Automatische Testdatengenerierung zur Absicherung fehlerbehafteter reaktiver Automatisierungssysteme

Dissertation

zur Erlangung des Doktorgrades
der Naturwissenschaften

vorgelegt von

Benjamin Kormann

geboren am 19.11.1980 in Marktredwitz

genehmigt von der

Fakultät für Mathematik/Informatik und Maschinenbau

der Technischen Universität Clausthal

Tag der mündlichen Prüfung

13.09.2016

Vorsitzender der Promotionskommission

Prof. Dr. rer. nat. Jörg P. Müller

Betreuer

Prof. Dr. rer. nat. Christian Siemers

Gutachter

Prof. Dr. rer. nat. Andreas Rausch, Prof. Dr.-Ing. Frank Schiller

Vorwort

Die vorliegende Dissertation stellt das Resultat meiner wissenschaftlichen Arbeit am Institut für Prozess- und Produktionstechnik der Technischen Universität Clausthal und am Lehrstuhl AIS (ehemals itm) der TU München dar. Meine Arbeit an der Dissertation wurde dabei stets von meinem Arbeitgeber, der ITQ GmbH, unterstützt.

Vorrangig gilt mein Dank meinem Doktorvater Herrn Prof. Dr. Christian Siemers für die fachliche Betreuung und die Möglichkeit, an der TU Clausthal zu promovieren. In ganz besonderer Erinnerung halte ich die sehr konstruktiven Gespräche in angenehmer Atmosphäre. Ich freue mich auf die weitere Zusammenarbeit. Des Weiteren bin ich Herrn Prof. Dr.-Ing. Frank Schiller sehr verbunden, der durch sein uneigennütziges Engagement ganz wesentlich zum Erfolg beigetragen hat.

Ein weiteres Dankeschön gilt ehemaligen Kollegen für die zahlreichen Fachdiskussionen. Diese haben sowohl die fachliche als auch persönliche Weiterbildung gefördert. Daraus sind auch private Freundschaften entstanden. Hier möchte ich meine ehemaligen Bürokollegen Dr.-Ing. Steven Braun, Dr.-Ing. Markus Friedrich und den im Geiste verbundenen Informatiker-Kollegen Dr. Martin Früchtl ganz besonders hervorheben. Es bedeutet mir sehr viel, in den Freundeskreis eines Jeden von Euch aufgenommen worden zu sein.

Meinen Eltern kann ich wohl nie hoch genug danken. Deren Einstellung zu Bildung, lebenslangem Lernen und einem perfektem Verhältnis aus Fördern und Fordern haben meinen Lebensweg überhaupt erst möglich gemacht. Dafür danke ich Euch von Herzen und werde versuchen, diese Lebenseinstellung bestmöglich weiterzugeben.

Zuletzt gilt mein Dank meiner lieben Frau Katrin für ihren bedingungslosen Rückhalt und ihrem grenzenlosen Verständnis für die zahlreichen Abende und Wochenenden, an denen ich sie vertrösten musste. Ich freue mich auf die weitere gemeinsame Zeit mit Dir und unserem Sohn.

Ihrlerstein, im November 2016
Benjamin Kormann

Kurzfassung

Durch den steigenden Funktionsumfang moderner Maschinen und Anlagen erhöht sich insbesondere die Komplexität der Steuerungssoftware. Der Test im Allgemeinen und der SPS Steuerungssoftware im Speziellen, wird im überwiegend Mechanik dominierten Maschinen- und Anlagenbau, noch nicht im vollen Umfang eingesetzt. Eine wesentliche Aufgabe entfällt dabei auf die Ermittlung der Testparameter, d.h. der Bestimmung von Eingabedaten eines Testfalls zur Stimulation des Testobjekts. Gängige Verfahren basieren meist auf Spezifikationen oder Funktionsbeschreibungen. Mechatronische Systeme haben einen Komplexitätsgrad erreicht, dass eine detaillierte Spezifikation kaum möglich ist. Im Maschinen- und Anlagenbau liegen u.a. auch aus zeitlichen Gründen häufig keine entsprechenden Spezifikationen vor.

Da das Laufzeitverhalten von reaktiven Systemen stark von der Umgebung, d.h. dem technischen Prozess und physikalischen Gesetzmäßigkeiten bestimmt wird, müssen diese Systeme besonders gegenüber auftretenden Fehlern robust sein. Der Anteil von Fehlerbehandlungsroutinen in einer modernen Steuerungssoftware ist bereits heute größer als 50 %, gemessen an der Gesamtkomplexität der Software. Diese Funktionalität bzw. die Softwarebausteine, die im Fehlerfall ausgeführt werden und für die Stabilität und Qualität der Maschine bzw. Anlage entscheidend sind, werden bislang im durchgeführten Test jedoch nicht im entsprechenden Umfang berücksichtigt.

In dieser Arbeit wurde ein Verfahren zur zeitdiskreten Testdatenermittlung von SPS Softwarebausteine entworfen, die sich in Interaktion mit potentiell fehlerhaften technischen Komponenten befinden. Dazu wird im Rahmen der modellbasierten Systementwicklung ein auf SysML basiertes System- und Fehlermodell entwickelt, welches als Grundlage für die Testdatengenerierung von zyklischen IEC 61131-3 Softwarebausteinen eingesetzt wird.

Der Kern des Verfahrens ist die Ermittlung der Testeingabedaten als mehrstufiges Constraint Satisfaction Problem (CSP), kombiniert mit symbolischer Ausführung. Dazu wird der Ausgangsquelltext in eine Zwischenrepräsentation überführt, aus der die Artefakte für die Testdatengenerierung erzeugt werden. Unter Ausnutzung der Datenabhängigkeit zyklisch ausgeführter Bausteine wird der erweiterte Kontrollflussgraph (Zwischenrepräsentation) ausgerollt. Dadurch ergibt sich eine endliche Anzahl an zu lösenden Constraint Satisfaction Problemen, wodurch der Pfad und somit die Eingabedaten zur Stimulation des Testobjekts berechnet werden kann. Die Auswahl des Testkriteriums wird unter der Einbeziehung der Fehlermodellinformationen und einer bidirektionalen Kopplung mit dem zu analysierenden IEC 61131-3 Quelltext vorgenommen. Das erarbeitete Verfahren wurde an zwei repräsentativen Applikationsbeispielen evaluiert.

Man kann ein Problem nicht mit der Denkweise lösen, die es erschaffen hat.
(Albert Einstein)

Inhaltsverzeichnis

Abkürzungsverzeichnis

AS	Ablaufsprache
AWL	Anweisungsliste
BD	Blockdiagramm
CAD	Computer Aided Design
CoDeSys	Controller Development System
CSP	Constraint Satisfaction Problem
E/A	Ein-/Ausgabe
EVA	Eingabe Verarbeitung Ausgabe
FUP	Funktionsplan
HDL	Hardware Description Language
HIL	Hardware-in-the-loop
HMI	Human Machine Interface
IBD	Internes Blockdiagramm
IEC	International Electrotechnical Commission
I/O	Input / Output
KOP	Kontaktplan
LOC	Lines of Code
MIL	Model-in-the-loop
MIT	Massachusetts Institute of Technology
OCL	Object Constraint Language
OEM	Original Equipment Manufacturer
OMG	Object Management Group
PAA	Prozessabbild der Ausgänge
PAE	Prozessabbild der Eingänge
PIL	Processor-in-the-loop
POE	Programmorganisationseinheit
SIL	Software-in-the-loop
SPS	Speicherprogrammierbare Steuerung
ST	Strukturierter Text
SysML	Systems Modeling Language
TPT	Time Partition Testing
UML	Unified Modeling Language
USV	Unterbrechungsfreie Stromversorgung
VDMA	Verband Deutscher Maschinen-/Anlagenbau e.V.
VPS	Verbindungsprogrammierte Steuerung

KAPITEL 1

Einführung und Überblick

In der vorliegenden Arbeit wird ein Verfahren zur automatischen Testdatengenerierung von Steuerungssoftware vorgestellt. Auf Basis eines Fehlermodells werden einerseits die Menge der Testeingabedaten reduziert und andererseits die Aussagekraft durch den inhärenten Bezug zu Fehlern erhöht. Es erfolgt eine Fokussierung auf die Bedürfnisse des Maschinen- und Anlagenbaus, insbesondere den spezifischen Eigenschaften zyklisch ausgeführter IEC 61131-3 Softwarebausteine.

1.1 Kontext der Arbeit

Zur Einordnung dieser Arbeit wird zunächst das Umfeld, in das sich die Arbeit einbettet näher erläutert. Darauf aufbauend wird eine Problemstellung abgeleitet, die die Grundlage für die anschließende Problemanalyse und die detaillierte Aufgabenstellung in Kapitel 2 darstellt.

1.1.1 Hintergrund und Motivation

Der Maschinen- und Anlagenbau war in seinen Anfängen eine vollständig aus den Herausforderungen der Mechanik getriebene Disziplin. Vor über 50 Jahren erweiterte sich der Maschinen- und Anlagenbau mit den ersten Hardwarekomponenten um die Disziplin der Elektrotechnik, was eine höhere Flexibilität durch Steuerungs- und Regelungsaufgaben ermöglichte. Mit der Einführung speicherprogrammierbarer Steuerungen in den 70er und 80er Jahren hielt die junge Disziplin der Informatik Einzug in den Maschinen- und Anlagenbau und eröffnete eine hardwareunabhängige Funktionsrealisierung in Form von flexibel miteinander kombinierbaren Programmbausteinen [Zan06]. Der Automatisierungsgrad moderner Produktionssysteme erhöht sich kontinuierlich [REG13]. Die führt letztendlich dazu, dass die Disziplin der Softwareentwicklung eine zentrale Rolle einnimmt [Rus13], wodurch diese zum dominierenden Element für Innovationen wird.

Durch die zunehmende Komplexität erfährt die ganzheitliche Betrachtung der Systementwicklung, das sog. Systems Engineering, eine zentrale Bedeutung. Dabei muss eine kontinuierliche, vollumfängliche Einbeziehung aller Disziplinen berücksichtigt werden. Aufgrund des steigenden Funktionsumfangs moderner Maschinen und

Anlagen werden vermehrt modellbasierte Ansätze zur Beherrschung der Komplexität eingesetzt [VHSFL14], [BKFVH14]. Zur Sicherstellung und Dokumentation der geforderten Funktionalität stellt der Qualitätsnachweis demnach in zunehmendem Maße eine zentrale Phase des Entwicklungsprozesses dar [Ber07]. Der wesentliche Fokus bei der Entwicklung einer Maschine oder Anlage liegt in der Realisierung des sog. Geradeauslaufs, d.h. der primären Funktionalität. Aus zeitlichen Gründen und einem hohen Kostendruck wird deshalb das Hauptaugenmerk auf die Verifikation der Primärfunktionalität gelegt, um den Kundenwunsch sicherzustellen [KTVH12]. Bei zeitlichen Engpässen in der Entwicklung werden häufig Kürzungen in den letzten Phasen, d.h. überwiegend in der Verifikation vorgenommen.

Da das Laufzeitverhalten von reaktiven Systemen stark von der Umgebung, d.h. dem technischen Prozess und physikalischen Gesetzmäßigkeiten bestimmt wird, müssen diese Systeme besonders gegenüber auftretenden Fehlern robust sein. Der Anteil von Fehlerbehandlungsroutinen in Softwarekomponenten ist bereits heute größer als 50 %, gemessen an der Gesamtkomplexität der Software [Mon01]. Diese Funktionalität bzw. die Softwarebausteine, die im Fehlerfall ausgeführt werden und für die Stabilität und Qualität der Maschine oder Anlage entscheidend sind, werden im durchgeführten Test jedoch nicht im entsprechenden Umfang berücksichtigt, d.h. sie werden nicht oder ohne Systematik getestet [KTVH12]. Durch die im Vergleich zur Automobilindustrie üblicherweise geringen Stückzahlen im Maschinen- und Anlagenbau fließen Aufwände des Engineerings in einem hohem Maße in den Gesamtaufwand. Aus diesem Grund, bedarf es effektiver Ansätze, die einen reduzierten Testdatensatz hoher Aussagekraft bei geringem, gegebenem Aufwand ermöglichen. Durch hohe äußere Einflüsse stellen insbesondere Fehlersituationen eine zentrale Herausforderung dar. Meist werden diese durch die Steuerungssoftware nicht ausreichend berücksichtigt bzw. deren umgesetzte Mechanismen nicht umfassend getestet. Die dazu notwendigen Testeingabedaten können aus Gründen mangelnder Zeit, Methodik sowie fehlender Werkzeugunterstützung zur Automatisierung nicht ohne entsprechenden Aufwand abgeleitet werden [KFVH12].

Der Verifikation aktueller Systeme, wird eine zunehmende Bedeutung zuteil. Durch zusätzliche Anforderungen, die aufgrund von verschiedensten Normen die Entwicklung maßgeblich beeinflussen und zusätzliche flankierende Maßnahmen, die die Systemkomplexität erhöhen, führen zu einem umfangreichen Design. Damit einhergehend werden Verifikationsmethoden zum dokumentierten Nachweis der Funktionserfüllung, unter erhöhten Rahmenbedingungen, d.h. durch Forderungen in Normen unerlässlich [ISO09], [IEC02], [IEC13].

Um diese Forderung zu erfüllen werden unter anderem formale Verifikationsmethoden eingesetzt. Dadurch wird es möglich, den Erfüllungsgrad von Eigenschaften eines Systems formal, d.h. über mathematische Modelle und Verifikationskriterien, nachzuweisen. In der Forschung werden formale Methoden im Maschinen- und Anlagenbau vermehrt erprobt [SF12], [KWV10], [War09], [WVJF06], [Gre07], [VHFH12] und deren Einsatzpotential optimiert. Eine wesentliche Eigenschaft der formalen Verifikation ist der schnell ansteigende Parameterraum, so dass die Anwendbarkeit meist erst durch problembezogene Abstraktionen gegeben ist.

Der Software- und Systemtest sind bei weitem das wichtigste Maß zum Qualitätsnachweis [PJR09]. Nach Dijkstra kann dieser zwar nicht die Abwesenheit von Fehlern nachweisen, jedoch kann durch eine repräsentative Menge von Testfällen, das

Konfidenzmaß der Fehlerfreiheit[1] erhöht werden. Eine wesentliche Aufgabe entfällt dabei auf die Ermittlung der Testparameter, d.h. der Bestimmung von Eingabedaten eines Testfalls zur Stimulation des Testobjekts. Manuelle Verfahren basieren meist auf der Ableitung von Testdaten aus prosaischen Spezifikationen oder Funktionsbeschreibungen und zeichnen sich meist durch ihre hohe Zeitintensivität aus. Da durch die geringen Stückzahlen im Maschinen- und Anlagenbau ein besonderer Fokus auf zeiteffiziente Mechanismen gelegt wird, sind manuelle Verfahren selten praktikabel. Deshalb wurden Verfahren entwickelt, die durch Transformationen und Explorationen von Entwicklungsartefakten automatisch Testdaten für den Verifikationsprozess generieren [McM04], [Lin08]. Linder generiert Testdaten aus vorhandenen Signalflussplänen und verweist auf die Notwendigkeit der Generierung aus codebasierten Testverfahren [Lin08]. Der Test im Allgemeinen und der SPS Steuerungssoftware im Speziellen wird im überwiegend Mechanik dominierten Maschinen- und Anlagenbau noch nicht im vollen Umfang eingesetzt [KTVH12].

1.1.2 Problemstellung und Ziele

Der zum Test eines (Software-)Systems zugrundeliegende Parameterraum ist von derart großem Umfang, dass ein vollständiger Test im Sinne der Prüfung aller möglichen Belegungen praktisch nicht realisierbar ist. Im schlechtesten Fall ist der Parameterraum unendlich groß, so dass auch theoretisch keine vollständige Verifikation möglich ist. Das wesentliche Ziel besteht demnach darin, eine geeignete Menge an Testparameterbelegungen zu identifizieren, die in endlicher Zeit überprüfbar sind und die im System manifestierte Fehler aufdeckt, so dass diese anschließend behoben werden können [Die09].

Der Prozess, die richtigen Testdaten zu finden, entspricht i.a. einem Ratevorgang, der durch Qualifikation, Erfahrung, Systemkomplexität und der zur Verfügung stehenden Zeit kontrolliert wird. Reaktive Systeme zeichnen sich insbesondere durch ihre physikalische Beeinflussung der Komponenten aus der Umwelt aus. Aus diesem Wirkungsgefüge heraus entstehen situative Systembeeinflussungen, die einen Fehler in jeder einzelnen beteiligten Komponente herbeiführen können. Nachdem sich Fehler im System manifestiert haben, können deren Auswirkungen einerseits weitere Fehler nach sich ziehen und andererseits zu Systemanomalien führen. Ein wesentliches Ziel der Entwicklung reaktiver Systeme ist es, Maßnahmen zu ergreifen, um derartige Fehlerfälle frühzeitig, d.h. während der Entwicklung zu identifizieren und eine Erhöhung der Absicherung des Systems bei eventuellem Fehlereintritt zu erzielen. Der Softwaretest repräsentiert ein vielfältiges Forschungsgebiet [LG10], [BDJ07], [UL07], [Ham13], [RTSVH14], [RRTVH14], [USVH14].

Mit der vorliegenden Arbeit wird das Ziel verfolgt, eine automatische Testdatengenerierung für zyklische IEC 61131-3 basierte Steuerungssoftware, auf Grundlage von in der Entwurfsphase des technischen Systems berücksichtigten Komponentenfehlern, zu entwickeln. Die generierten Testdaten dienen in der Testumgebung als Stimuli für das Testobjekt. Dieser Lösungsansatz adressiert und konkretisiert den Bedarf für

1 Es wird i.a. angenommen, dass 3 Fehler in 1000 Zeilen Quelltext enthalten sind.

fehlerbasierte Tests in der Produktionsautomation [RUPVH15]. Verfügbare Generierungsverfahren erzeugen zumeist eine zu hohe Menge an Testfällen bzw. Testdaten, die sich häufig auch durch eine zu geringe Aussagekraft auszeichnen. Des Weiteren wird die zyklische Ausführungslogik von IEC 61131-3 basierten Softwarebausteinen nicht berücksichtigt, wodurch keine vollständige Generierung der Testeingabedaten möglich ist.

Diese Arbeit leistet somit einen wesentlichen Beitrag zu den folgenden Aspekten:

- Explizite Integration von Fehlern mechanischer Komponenten bei der Modellierung des technischen Automatisierungssystems.
- Automatische Generierung der Stimuli für den Test von zyklisch ausgeführten IEC 61131-3 basierten Applikationen.

1.2 Gliederung der Arbeit

Die Arbeit ist wie folgt aufgebaut: In Kapitel 2 erfolgt eine systematische Problemanalyse des Softwaretests reaktiver Systeme, woraus der Handlungsbedarf ermittelt und Anforderungen an eine Lösung abgeleitet werden. Im anschließenden Kapitel 3, dem Stand der Technik werden aus der Problemanalyse, Kriterien zur Bewertung existierender Ansätze zur Fehlermodellierung und Testdatengenerierung abgeleitet. Das Konzept in Kapitel 4 beschreibt alle Bestandteile eines Testdatengenerierungsverfahrens auf Basis eines Fehlermodells und adressiert dabei die an eine Lösung erhobenen Anforderungen und spezifischen Kriterien. Zur Bewertung des Konzepts wird eine prototypische Realisierung in Kapitel 5 beschrieben, die der anschließenden Evaluation in Kapitel 6 dient. Die für den quantitativen Vergleich der Testdatengenerierungsverfahren verwendeten Beispielprogramme sind im Anhang A aufgelistet. Abschließend erfolgt in Kapitel 7 eine Zusammenfassung des wissenschaftlichen und technischen Beitrags der gesamten Arbeit und ein Ausblick.

Der Aufbau der Arbeit ist in Abbildung 1.1 zusätzlich als kartographische Übersicht dargestellt.

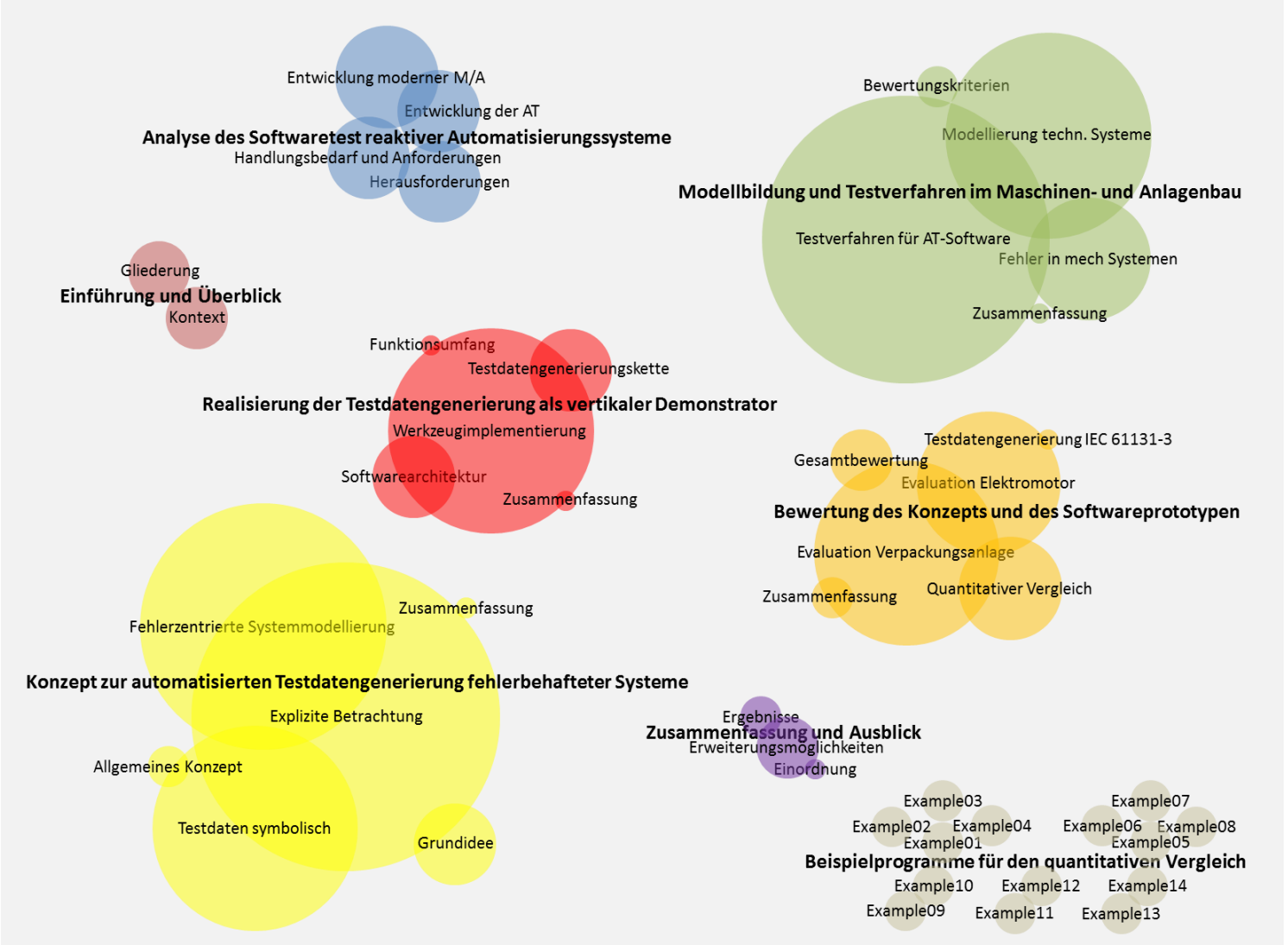


Abb. 1.1: Kartographische Übersicht der Arbeit

KAPITEL 2

Analyse des Softwaretests reaktiver Automatisierungssysteme

Durch die stetige Zunahme der Softwareanteile im Gesamtsystem moderner Maschinen und Anlagen nimmt die Absicherung der Softwarefunktionalität eine zentrale Rolle ein. Aufgrund der im Allgemeinen geringen Stückzahlen von Maschinen und Anlagen, schlagen die Kosten fürs Engineering besonders hoch zu Buche, weshalb ein strukturierter Test meist nicht umgesetzt werden kann. Besondere Herausforderungen stellen Fehlersituationen dar, die durch die Steuerungssoftware nicht ausreichend berücksichtigt bzw. deren umgesetzte Mechanismen nicht umfassend getestet wurden. Die dazu notwendigen Testeingabedaten können aus Gründen wie zum Beispiel mangelnder Zeit, Methodik sowie fehlender Werkzeugunterstützung zur Automatisierung nicht aufwandsarm abgeleitet werden.

Inhaltsverzeichnis

2.1 Entwicklung der Automatisierungstechnik

Die Funktionsrealisierung moderner Maschinen bzw. Anlagen wird in weiten Teilen durch die Steuerungssoftware erbracht. Der VDMA hat in seiner Zukunftsprognose die Verschiebung der Anteile zwischen den Disziplinen zwischen 1970 und 2020 skizziert, siehe Abbildung 2.1. Diese Entwicklung führt dazu, dass die Qualität der Steuerungssoftware einen entscheidenden Einfluss auf die Gesamtqualität und Stabilität einer Maschine oder Anlage ausübt. Die Herausforderungen der kommenden Jahre werden die Bewältigung der Gesamtkomplexität bei steigender Bedeutung der Software sein. In dieser Phase wird es besonders auf effiziente Prozesse, Methoden und Werkzeuge des Softwaretests zum Nachweis der Funktionserfüllung ankommen.

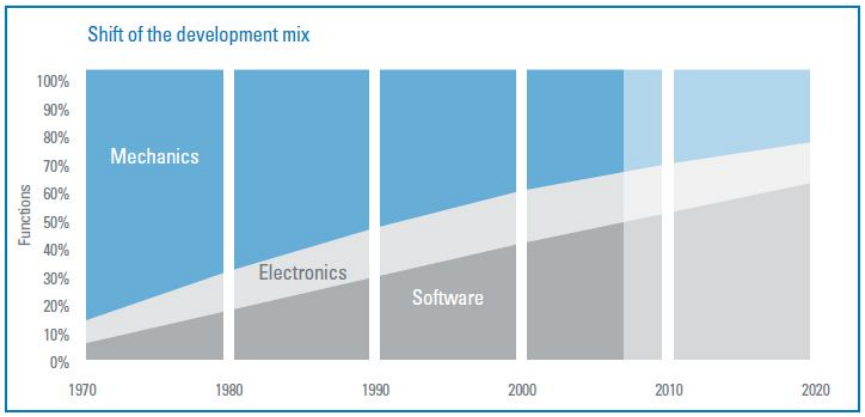


Abb. 2.1: VDMA Zukunftsprognose [ITQ11]

Die historischen Hintergrundinformationen zur Entwicklung der SPS und ihren Programmiersprachen in diesem Kapitel sind aus den Beschreibungen in [Zan06] zusammengefasst.

2.1.1 Von der VPS zur SPS

Der Maschinen- und Anlagenbau war in seinen Anfängen eine vollständig aus den Herausforderungen der Mechanik getriebene Disziplin. Vor über 50 Jahren erweiterte sich der Maschinen- und Anlagenbau mit den ersten Hardwarekomponenten um die Disziplin der Elektrotechnik, was eine höhere Flexibilität durch Steuerungs- und Regelungsaufgaben ermöglichte. Die Steuerung von Anlagen erfolgte in den Anfängen durch logische Verbindungen der Einzelkomponenten mit Hilfe von Relais und Schütze. Zur Steuerung einer Anlage genügten logische Operationen wie UND, ODER, NOR, Flipflops und Kippschaltungen für zeitliche Aspekte. Transistoren ersetzten schließlich nach und nach die vorhandenen Relais. Da sie aufgrund ihrer Konstruktion Bauteile ohne bewegliche Kontakte sind, wurden durch den Fortschritt des Transistors bereits um 1960 Anwendungen möglich, zu deren Arbeitsgeschwindigkeit Relais nicht in der Lage gewesen wären.

Mitte der 60er Jahre entstand ein hierarchisches Konzept mit dem Ziel, die Zuverlässigkeit von Kraftwerken zu erhöhen. Daraus entwickelte sich die bis heute eingesetzte Automatisierungspyramide. Zu dieser Zeit kamen schnellere, integrierte Logikbausteine

auf den Markt. Für die Anwendbarkeit im Maschinen- und Anlagenbau mussten diese Anforderungen, wie die sog. stör- und zerstörsichere Logik erfüllen. Mit der Einführung speicherprogrammierbarer Steuerungen in den 70er und 80er Jahren hielt die junge Disziplin der Informatik Einzug in den Maschinen- und Anlagenbau und eröffnete eine hardwareunabhängige Funktionsrealisierung in Form von flexibel miteinander kombinierbaren Programmbausteinen. Die bis dahin übliche Realisierung durch Verdrahtung von Flachbaugruppen wurde von nun an auch verbindungs- bzw. verdrahtungsprogrammierte Steuerung (VPS) genannt, um den Unterschied zur speicherprogrammierbaren Steuerung (SPS) zu verdeutlichen. Definition 1 erläutert den Einsatz und die Aufgaben einer SPS.

Steuerungen konnten anfänglich lediglich programmiert werden und waren aufgrund der noch zu teuren Speichermedien nicht speicherprogrammierbar. Während dieser Übergangszeit wurden die ersten speicherprogrammierbaren Steuerungen weiterhin individuell projektiert und verdrahtet. Dabei wurden die binären Eingänge logisch verknüpft und anschließend die Ausgänge entsprechend geschaltet.

Abbildung 2.2 zeigt den konzeptionellen Aufbau einer speicherprogrammierbaren Steuerung. Eine SPS folgt dem klassischen EVA-Prinzip (Eingabe, Verarbeitung, Ausgabe) der Datenverarbeitung. Zu Beginn werden alle an den Eingängen anliegenden Signalzustände eingelesen und dadurch das Prozessabbild der Eingänge (PAE) erneuert. Das Steuerwerk bringt das geladene Programm zur Ausführung, indem alle darin enthaltenen Anweisungen sequentiell abgearbeitet werden. Dadurch entsteht ein neues, aktuelles Prozessabbild der Ausgänge (PAA), das anschließend auf die Ausgangsregister kopiert wird. Dieser Vorgang erfolgt zyklisch, d.h. dieser Vorgang vom Erneuern des Prozessabbilds bis zum Neusetzen des Abbilds auf die Ausgänge läuft in einer Endlosschleife ab. Die Zeit, die für einen Durchlauf benötigt wird, muss nicht immer konstant sein, da sich abhängig von den Eingangswerten unterschiedliche Anweisungen des Programmes ergeben können. Jedoch läuft dieser Zyklus in einer fest vorgegebenen Zeitscheibe, der sog. Zykluszeit ab, die genau so gewählt wird, dass alle auszuführenden Anweisungen darin ausgeführt werden können. Eventuell auftretende Differenzzeiten zwischen der reinen Abarbeitungszeit und der Zykluszeit werden durch aktives Warten aufgefüllt.

Definition 1 (*Speicherprogrammierbare Steuerung nach [DIN04]*)

> *„Ein digital arbeitendes elektronisches System für den Einsatz in industrieller Umgebung mit einem programmierbaren Speicher zur internen Speicherung der anwenderorientierten Steuerungsanweisungen zur Implementierung spezifischer Funktionen wie z.B. Verknüpfungssteuerung, Ablaufsteuerung, Zeit-, Zählfunktion und arithmetische Funktionen, um durch digitale oder analoge Eingangs- und Ausgangssignale verschiedene Arten von Maschinen oder Prozessen zu steuern.“*

2.1.2 Die Sprachen der IEC 61131-3

Der Maschinen- und Anlagenbau in Deutschland ist geprägt durch sehr gut ausgebildete Techniker, wie Elektriker oder Monteure. Beim Übergang der Steuerungstechnik von VPS zur SPS wurde es notwendig, dass die zur Funktionserfüllung der Anlagen notwendige Steuerungssoftware durch die Techniker erstellt, erweitert und auch

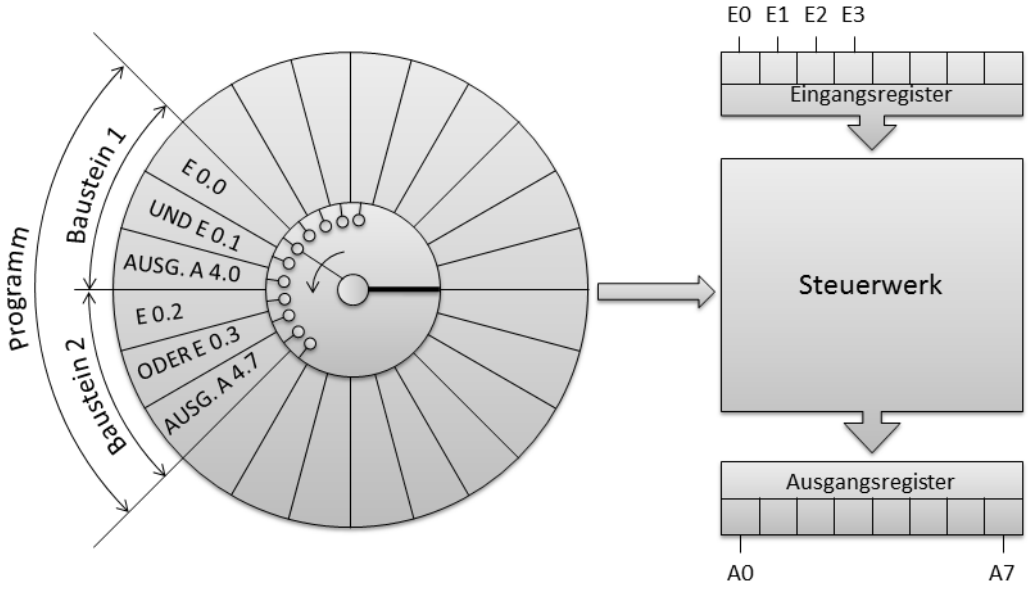


Abb. 2.2: Der konzeptionelle Aufbau einer SPS [Zan06]

gewartet werden kann. Der primäre Ausbildungsinhalt bestand jedoch nicht im Erlernen höherer Programmiersprachen wie bspw. Pascal. Damit der Technologiesprung zur SPS machbar wurde, war es notwendig, die Programmierung stark an die Bedürfnisse der Techniker mit deren Kenntnisstand anzupassen. Deshalb wurden zu Beginn die folgenden drei Sprachen entworfen:

- Kontaktplan (KOP): Diese Art der Beschreibung besitzt den stärksten Bezug zu den Relais-Schaltungen, da es dem Stromlaufplan ähnelt. Durch den vermehrten Einsatz dieser Sprache in der Automobilindustrie stellt KOP eine der wichtigsten Sprachen der Programmiernorm dar.
- Funktionsplan (FUP): Mit Hilfe dieser Sprache werden besonders primitive logische Verknüpfungen zwischen Eingangswerten dargestellt. Der Funktionsplan wird am häufigsten in der Verfahrenstechnik eingesetzt.
- Anweisungsliste (AWL): Sie ähnelt der Assemblersprache und besteht aus einer mnemonischen Sprache. Die Lesbarkeit und das Verständnis der Funktionalität mit dieser Sprache ist für komplexe Zusammenhänge nicht uneingeschränkt geeignet.

Diese Sprachen wurden in der Norm IEC 61131-3 [IEC03a] spezifiziert, die noch um die Sprachen Ablaufsprache (AS) und Strukturierter Text (ST) erweitert wurde. Die Ablaufsprache ist die insgesamt dritte graphische Sprache der Programmiernorm, hat einen starken Bezug zur schrittweisen Beschreibung von Prozessen und ermöglicht eine Anordnung der einzelnen Abfolgen. Neben AWL existiert mit der Sprache ST eine weitere textuelle Programmiersprache, die syntaktisch Pascal ähnelt und eine Anlagenprogrammierung auf einem höheren Abstraktionsniveau ermöglicht. In den gängigen Programmierumgebungen können die Programme aus jeder IEC Sprache

in jede weitere IEC Sprache ohne Funktionalitätsverlust transformiert werden. Das Ziel der IEC 61131-3 war es, Anwendersoftware herstellerunabhängig und damit portierbar zu erstellen. Da in den heutigen Programmierumgebungen der IEC 61131-3 Norm teilweise Unterschiede existieren, ist dieses Ziel nicht vollständig erfüllt worden. Migrationen von einer Steuerungsplattform auf die eines anderen Herstellers stellt heutzutage ein noch immer schwieriges Unterfangen dar. Darauf wird bereits in der IEC 61131-8 [IEC03b] hingewiesen mit:

> „This facilitates the reusability of control software designs for different controller types, even though some effort will almost always be required in order to move control programs from one controller family to another."

Die Tabellen 2.1 und 2.2 zeigen Beispielprogramme zu jeder der fünf IEC 61131-3 Sprachen [Bec12]. Die Speicherprogrammierbare Steuerung hat sich zur Steuerung industrieller Maschinen bzw. Anlagen im Vergleich zum Personal Computer (PC) aus mehreren Gründen durchgesetzt. Die Anforderungen an PCs umfassen üblicherweise eine stets höhere Performance zu günstigen Preisen, wohingegen eine Steuerung im industriellen Einsatz hohen Temperaturbedingungen u.a. auch Temperaturschwankungen, Luftfeuchtigkeit etc. ausgesetzt sind und aus diesem Grund eine hohe Robustheit vorweisen müssen. Das zuverlässige Wiederanlaufverhalten ist bei gängigen PCs jedoch nicht immer gegeben. Ein weiterer elementarer Vorteil von speicherprogrammierbaren Steuerungen besteht im geringeren Aufwand im Engineering, was bei den üblicherweise relativ geringen Stückzahlen einen wesentlichen Vorteil darstellt. An eine SPS angeschlossene E/A-Baugruppen werden für den Programmierer derart abstrahiert, so dass die angeschlossene Peripherie, nach der entsprechenden

Tab. 2.1: Beispielprogramme zu den textuellen IEC 61131-3 Sprachen

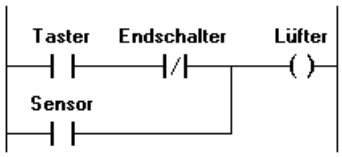
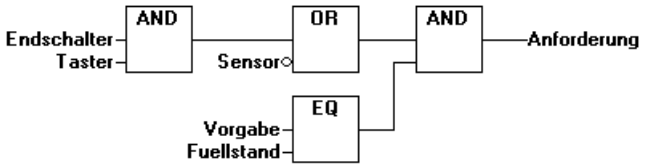
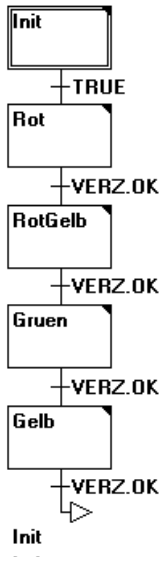
Programmiersprache	Beispielprogramm

Anweisungsliste	`Start:  LD Beckenfuellstand`
	`  GE 13`
	`  JMPC Pumpe_ein`
	`  R Pumpe_Ansteuerung`
	`  JMP Ende`
	`Pumpe_ein:  S Pumpe_Ansteuerung`
	`Ende:`

Strukturierter Text	`CASE Temperatur_Ofen OF`
	`  60..99:  Heizung := 80;`
	`  100..149:  Heizung := 60;`
	`  150..199:  Heizung := 35;`
	`  200..250:  Heizung := 10;`
	`ELSE`
	`  Alarm := TRUE;`
	`END_CASE;`

Konfiguration, über einen direkten Zugriff auf die Ein- und Ausgabevariablen der SPS angesprochen werden können.

Tab. 2.2: Beispielprogramme zu den graphischen IEC 61131-3 Sprachen

Programmiersprache	Beispielprogramm
Kontaktplan	
Funktionsplan	
Ablaufsprache	

2.2 Entwicklung moderner Maschinen und Anlagen

In diesem Abschnitt werden der Ist-Stand und Herausforderungen bei der Entwicklung moderner Maschinen und Anlagen, insbesondere im Kontext der Steuerungsapplikation präsentiert. Auf dieser Basis werden Anforderungen an eine Lösung zur fehlerorientierten Testdatengenerierung erhoben.

2.2.1 Ist-Stand der Entwicklung und Anforderungserhebung

An einer Umfrage haben zwei Unternehmen aus der Luft- und Raumfahrt, vier Unternehmen aus dem Bereich Automobil (OEM und Zulieferer) und zehn Unternehmen aus dem Maschinen- und Anlagenbau teilgenommen [KFVH12], [KTVH12]. Das angesetzte Ziel der Umfrage war, den Ist-Stand der Entwicklung in den oben genannten

Branchen zu erheben, um die unterschiedlichen Herangehensweisen und Problemfelder zu identifizieren und daraus den Handlungsbedarf speziell für den Maschinen- und Anlagenbau abzuleiten. Die erhobenen Daten enthalten die folgenden Aspekte: Anforderungserhebung- und verwaltung, Test, Simulation, Anforderungs- und Testspezifikation, Fehlerszenarien, Prozesse, Qualitätsmaße und eingesetzte Werkzeuge.

Tabelle 2.3 stellt eine kompakte Zusammenfassung der Umfrageergebnisse der Bereiche Luft- und Raumfahrt und Automobil dar. Diese belegen sehr deutlich, dass aufgrund der hohen Anforderungen an das Produkt und die jeweilige Zielgruppe umfangreiche Testmethoden und effiziente Testprozesse zur Bewältigung der Systemkomplexität eingeführt wurden. Gemessen am gesamten Projektvolumen kann der Testaufwand bis zu 80 % betragen. Dies ist vor allem durch die teilweise gesetzlich vorgeschriebenen Nachweise und gültigen Normen essentiell. Besonders wegen der hohen Stückzahlen sind derartige Maßnahmen notwendig und auch realisierbar.

Die Ergebnisse der Unternehmensbefragung aus dem Maschinen- und Anlagenbau sind in Tabelle 2.4 dargestellt. Es ist deutlich abzulesen, dass der Test nur unzureichend unterstützt wird. Im Vergleich zu den anderen beiden Branchen, in denen automatische Testprozesse und moderne Testmethoden eingesetzt werden, erfolgt der Test des Steuerungssystems im Maschinen- und Anlagenbau meist nur auf Basis manueller Entwicklertests. Dies ist darauf zurückzuführen, dass wegen der geringen Stückzahlen der Test nicht als explizite Phase des Entwicklungsprozesses berücksichtigt wird, sondern als integraler Bestandteil der Softwarekonstruktion betrachtet wird. Hierbei verschärft sich die Situation, denn Entwicklertests besitzen nur eine sehr geringe Fehleraufdeckungsrate, da im Allgemeinen nicht die Erfüllung der geforderten Funktionalität, sondern die Umsetzung des Entwicklers getestet wird. Entwicklertests sind nicht ausreichend, um Fehler aufzudecken und qualitativ hochwertige Produkte zu entwickeln [PEN07]. Dadurch verschwimmen die Grenzen und systematische Tests können auch wegen mangelnder domänenspezifischer Testwerkzeuge nicht effizient durchgeführt werden. Ein erster Ansatz zur Beschreibung und automatischen Ausführung von Testfällen für IEC 61131-3 Software wird in [KTVH12] präsentiert.

Die zunehmende Komplexität der Maschinen bzw. Anlagen und dadurch auch der Steuerungssoftware verschärft jedoch die Notwendigkeit für zielgerichtete und zeiteffiziente Testverfahren zur Absicherung des Gesamtsystems. In der Umfrage wurde gezielt der relevante Testbedarf zur Bewältigung der Systemkomplexität abgefragt. Dabei wurde deutlich, dass neben der Überprüfung der geforderten Anlagenfunktionalität besonders jene Szenarien von sehr hoher Bedeutung sind, die sowohl die Maschine oder Anlage als auch das Personal gefährden können. Im speziellen wurden hier Fehlerszenarien, d.h. wahrscheinliche und bekannte Fehler hervorgehoben. Bislang werden aufgrund der fehlenden frühzeitigen, systematischen Testdurchführung Fehler häufig erst während der Inbetriebnahme erkannt [Dom07], [Wün07]. Die Kosten zur Behebung eines Fehlers steigen exponentiell mit der Verweildauer im System., Abbildung 2.3. Deshalb bedarf es Verfahren, die eine frühzeitige Fehlerlokalisierung und Fehlerbehebung unterstützen.

Tab. 2.3: Qualitative Umfrageergebnisse in Luft- und Raumfahrt und Automobil [KFVH12]

Aspekt		Luft- und Raumfahrt			Automobil			
Programmiersprache		ANSI C		ADA	Assembler	C/C++	Matlab/Simulink	
Softwarekomplexität		Kleine System < 30 k LOC		Komplexe Systeme > 200 k LOC	Kleine Systeme 50 k LOC		Komplexe Systeme 500 k LOC	
Entwicklungsprozessmodell		Funktionsorientiert		Iterativ	Modelbasiert (3-4 Wochen Iterationen)			
Reservierte Zeit fürs Testen		40-50 % des Gesamtaufwands		bis zu 80 % der Gesamtkosten	Softwaretest ca. 50 % der Projektlaufzeit		(Testzeit) / (Entwicklungszeit) = 2	
		Bis das System qualifizierbar fertig ist						
Art des Testens		Blackbox (Unit, System)		Whitebox (Unit, Modul)	Blackbox (SiL, HiL)		Whitebox (MiL, SiL)	
Dokumentierter Testnachweis		ESA Programme: bis zu 100 % Abdeckung		Unabhängige Softwareverifikation	Gesetz	Modellierungs-standards	Externe Audits	ASPICE
		DO178b	ECSS-E-ST-40C	Mil2167a	ISO 26262	KPI	OBD compliance	
Zeitpunkt des Testens		Ende der Implementierung		Nach jedem Bugfix und jeder Änderung	Nach jedem Zyklus	Nach jedem Change Request	Nach jedem geschlossenen Ticket	Permanent
					Nachdem die Funktion in das Steuergerät integriert		Parallele Test im Testfahrzeug	
Testabdeckung		Erfüllung der Anforderungen		Code Coverage	Erfüllung der Anforderung	Code Coverage	Zustands-übergänge	Code Review
Testressourcen		Unabhängiges Testteam	Developer (Unit Test)	Externe Tests (durch Supplier)	Entwickler (Anforderungsbasiertes Testen)		Testabteilung (System, Regression, Robustheit)	
Darstellen von Testfällen		Quelltext		Textuelle Beschreibung mit Diagrammen	Klassifikations-baum	Testschritte in Excel	XML	Quelltext
Ableitung von Testfällen		Anforderungsbasiertes Testen			Anforderungsbasiertes Testen (Äquivalenzklassen, Grenzwerte)			Modellbasiert via TPT
Entwicklung von Simulationsmodellen		Simulations- und Modellierungsabteilung			Entwickler	Zentralabteilung	Keine Simulation	
Werkzeug-unterstützung	Modellierung	keine Nennung			Matlab/Simulink / StateFlow			ASCET
	Versionsverwaltung	CVS		Clearcase / Clearquest	MKS	SVN	Vector eASEE	
	Bug Tracking	Clearcase / Clearquest		Trac	MKS	Jira	LOP	Inhouse Software
	Konfigurationsmanagement	keine Nennung			MKS	Clearcase	Vector eASEE	
	Testen	VectorCast/Cover		Java basierte Validierungsumgebung	CTE	EXACT	PolySpace	Inhouse Software
					TPT	LabCar Automation	Rational Test-Realtime	Target Simulation Module (dSpace)
	Deployment	keine Nennung			Inhouse Software			
	Anforderungen	Doors		UML Tools	Doors		Rational ClearQuest	

2.2.2 Handlungsbedarf für den Test der SPS Steuerungssoftware

Bei genauer Betrachtung der Umfrageergebnisse wird deutlich, dass der strukturierte Softwaretest über alle Entwicklungsphasen nicht gemäß dem möglichen Potential ausgeschöpft wird. Aktuell wird der Test als integraler Bestandteil der Softwareentwicklung betrachtet und zeichnet sich durch hohen manuellen Aufwand, intuitiv mögliche Fehlerszenarien abzuprüfen, aus. Dabei wird die Wahl der möglichen Situationen (Testeingabedaten) auf die Kreativität und die Erfahrung des Entwicklers und die ihm zur Verfügung stehende Zeit reduziert. Deshalb werden im Regelfall zufällig gewählte Parameterwerte manuell abgeleitet und damit das zu testende Objekt (Steuerungssoftware) stimuliert. Es bedarf objektiver Kriterien, auf deren Basis reproduzierbar, zielgerichtete Testdaten abgeleitet werden können.

Eine zentrale Herausforderung unter den Nennungen zeigt die fehlende Reaktion auf undefinierte Maschinenzustände als Ursache häufiger Fehler in der Steuerungssoftware. Diese undefinierten Maschinenzustände können in Folge einer mangelnden Prozessbeschreibung, aber auch als Konsequenz möglicher Defekte bzw. Verhaltensänderungen der eingesetzten Geräte (Sensoren, Aktoren) resultieren. Typische Testszenarien umfassen dabei Fehlfunktionen jeglicher Art sowie Maschinenfehler (bspw. plötzlicher Ausfall eines Sensors). Bei der Inbetriebnahme und während des Betriebs führen solche

Tab. 2.4: Qualitative Umfrageergebnisse im Maschinen- und Anlagenbau [KFVH12]

	Aspekt		Maschinen- und Anlagenbau			
Aktuelle Situation	Programmiersprachen		IEC 61131-3 (mixture)		C/C++	
	Softwarekomplexität		meisten nicht quantifizierbar		keine Metriken für graphische Sprache	
	Entwicklungsprozessmodell		keine Nennung			
	Reservierte Zeit fürs Testen		Keine explizite Zeit	als Teil der Softwarekonstruktion betrachtet		Bibliotheksfunktionen
	Werkzeug-unterstützung	Modellierung	Matlab / Simulink		COMOS	
		Versionsverwaltung	SVN		ENI	
		Bug Tracking	keine Nennung			
		Konfigurationsmanagement	keine Nennung			
		Testen	keine	Inhouse Software		
		Deployment	keine Nennung			
		Anforderungen	Microsoft Office			
	Art des Testens		hauptsächlich Funktionstests			
	Dokumentierter Testnachweis		Intern: Checklisten, Code Review	Tests in Form von Videoaufnahmen	allgemeine Performancetests	Not-Aus Tests
	Zeitpunkt des Testens		manueller Funktionstest während der Entwicklung		manuelle Interaktion mit Funktionen durch Wertemanipulation in der Visualisierung	
	Testabdeckung		Wird nicht erhoben, da nicht gefordert			
	Testressourcen		kein Testteam verfügbar	Verschiedene SPS (unterschiedliche Hersteller) stehen zur Verfügung		
	Darstellung von Testfällen		Beschreibung von Ablaufsequenzen		Zustandsmaschinen	Textuelle Beschreibung
	Entwicklung von Simulationsmodellen		aus Projektsicht nicht akzeptiert		nur akzeptiert, wenn Nutzen sichtbar	
	Informationen im Simulationsmodell		Benutzerinteraktion		Kollisionsinformationen (Roboter)	
			Zustand	Grenzwerte	Keine expliziten Simulationsmodelle vorhanden	
	Häufige Fehler in der Steuerungssoftware		fehlende Reaktion in undefinierten Maschinenzuständen durch unvollständige Prozessbeschreibung)			falsche I/O Adressen
			permanente Neuimplementierung von bereits implementierter Funktionalität			keine Standardisierung in Software
			falsches interlocking	negative Logik	fehlende Benachrichtigung auf HMI	
	Ableitung von Testfällen		Anforderungen	Spezifikation	Funktionsbeschreibung von HW/SW Modulen	
	QS Prozess zum Softwaretest		Kein standardisierter Prozess		Interne Code Reviews	
	Typische Testszenarien		Simulation von Maschinenfehlern	Fehlfunktion	Not-Aus	Projektabhängig
	Sind spezifizierte Testfälle ausführbar		Teilweise	Textuelle Beschreibung während Inbetriebnahme	Keine definierten Testfälle nach der Softwarekonstruktion	
Testbedarf	Testgranularität		Softwareelement (FB) -> Modul -> Maschine -> Anlage		Softwareelement während Entwicklung	Maschine und Module
	Relevanz Szenarien zu testen		sehr wichtig (Anlage wird komplexer)		Projektabhängig	
	Relevanz Fehlerszenarien zu testen		sehr wichtig (Klärung wahrscheinlicher Fehler)		wichtig (bekannte und wahrscheinliche Fehler)	
	Relevanz geforderte Funktionalität zu testen		sehr wichtig		wichtig	
	Welche Fehlersituation sollten getestet werden		bekannte und wahrscheinliche Fehlersituationen		bekannte Fehler mit gefährlichen Folgen	
	Unterstützung zur Fehlerdiagnose		Zustand und Trace der Software		vollständiges Bild der I/O	
	Relevanz Grenzwerte zu testen		sehr wichtig (Situationen, die Maschine oder Operator gefährden)			maximaler Durchsatz

Zusammenhänge zu Verzögerungen [Dom07] oder gar zu gefährlichen Situationen für Mensch und Maschine. Für eine langfristige Erhöhung der Zuverlässigkeit und eine frühzeitige Eliminierung möglicher Fehlerquellen bedarf es demnach eines Verfahrens, das die Beschreibung derartiger Fehlerfälle ermöglicht und das Systemverhalten in diesen Situationen explizit auf ausgewählte Aspekte hin getestet werden kann. In der Umfrage wurde dies als wesentlicher Bedarf erkannt.

Tabelle 2.5 illustriert den Handlungsbedarf für den Maschinen- und Anlagenbau bezogen auf die aus der Umfrage als wesentlich erachteten Kriterien in Abhängigkeit der jeweiligen Ausprägungen. Für einen effektiven Testprozess (Test Plan) sind neben der Testplanung (Test Procedure Specification, Test Case Specification) besonders die zielgerichtete Auswahl der testbezogenen Daten eine elementare Voraussetzung [IEE08]. Für den Test der Steuerungssoftware im Maschinen- und Anlagenbau sind besonders jene Situationen von besonderem Interesse, die aufgrund von Fehlern in den Komponenten der Anlage resultieren. Fehler dieser Art können unterschiedliche Ursachen haben und sowohl durch Wartungspersonal, als auch durch die Umgebung

(Korrosion) ausgelöst werden. Für eine bewertbare Testdurchführung bedarf es einer expliziten Berücksichtigung der relevanten Fehler bei der Modellbildung. Zur Unterstützung einer evolutionären Fehlermodellierung müssen sowohl grobgranulare als auch präzise Spezifikationen von Fehlern unterstützt werden. Darüber kann bei der Testdatengenerierung die entsprechende Testtiefe und somit die Absicherung des gesamten Systems skaliert werden. Die Notation zur Beschreibung des Fehlermodells muss für die Erreichung der Akzeptanz im Maschinen- und Anlagenbau graphisch erfolgen [HV05].

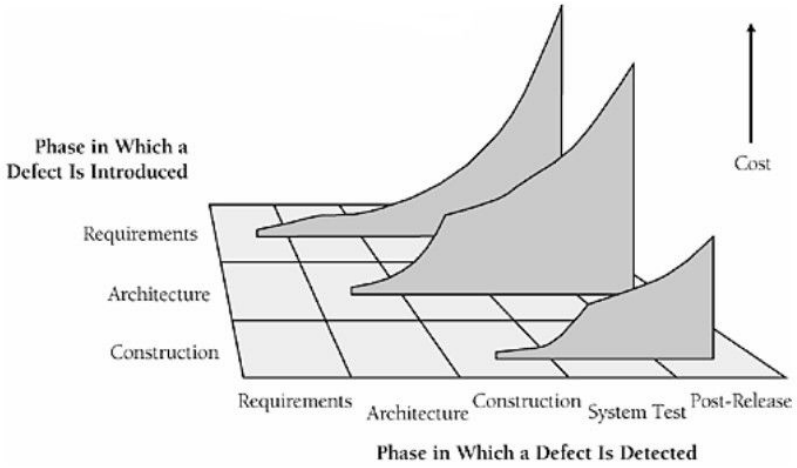


Abb. 2.3: Kostenabschätzung der Fehlerbehebung in Abhängigkeit der Fehlereinführung [McC09]

In der Luft- und Raumfahrt müssen zum Nachweis der Anforderungserfüllung neben den funktionalen Tests auch Strukturtests auf Komponenten- und Modulebene durchgeführt und auch nachgewiesen werden. Dies wird nicht zuletzt durch zahlreiche Normen, wie DO178b, ECSS-E-ST-40C oder Mil2167a gefordert. Im Vergleich zu den Funktionstests, die im Wesentlichen die gemäß der Spezifikation umgesetzten Funktionsmerkmale als Black-Box abprüfen, berücksichtigen White-Box Tests der Steuerungssoftware die tatsächlich vorhandenen Bestandteile des Quelltextes auf Kriterien, wie Erreichbarkeit, Abdeckung, Effizienz u.v.m. Die Betrachtung der Umsetzung ermöglicht eine präzise Analyse mit dem Ziel, Fehler bzw. mangelhafte Implementierungen frühzeitig zu identifizieren. Lammermann und Wappler [LW05] bezeichnen die Durchführung von Strukturtests als wichtigen Prozess zur frühzeitigen Erkennung von Fehlern und pflichten der damit verbundenen Generierung der Testdaten eine maßgebliche Rolle bei:

> „For the early detection of errors during software development, the white-box test, in addition to the function-oriented test, constitutes an important process. For this the generation of test data takes on a decisive role, since it determines the quality relevant input values for the test.“

Tab. 2.5: Zusammenfassung des Handlungsbedarfs im Maschinen- und Anlagenbau

Kriterium	**Ausprägung**	**Bewertung**	**Ausprägung**
Test	strukturiert	●———	intuitiv
Generierung Testdaten	automatisch	●———	manuell
Fehlermodell	explizit	●———	implizit
Modellierungssprache	graphisch	●———	textuell
Granularität	präzise	●———●	grob
Testverfahren	White-Box	●———●	Black-Box

2.2.3 Zusammenfassung

Durch die zunehmende Verschiebung der Funktionalität moderner Maschinen und Anlagen in die SPS Steuerungssoftware übernimmt diese auch die Funktionen zur Stabilität und Zuverlässigkeit des Gesamtsystems. Die Entwicklungshistorie der Automatisierungstechnik in Abschnitt 2.1 verdeutlicht dies. Die Tests zur Absicherung der Systemfunktionalität für den Geradeauslauf und im Fehlerfall, konnte in den Anfängen erst mit der Fertigstellung der Anlage vorgenommen werden. Erst durch die starke Durchdringung der Software ist es heutzutage möglich, Aktivitäten und Überprüfungen bereits in den frühen Phasen durchzuführen [Dom07], [Wün07]. Die Erkenntnisse aus den potentiellen Fehlern der Geräte in Maschinen bzw. Anlagen liegen bislang jedoch nicht in konkreten Modellen vor, so dass eine auf diese Aspekte gerichtete, frühzeitige Testdatengenerierung aktuell nicht möglich ist.

2.3 Herausforderungen beim fehlerorientierten Strukturtest

In der Softwareentwicklung bezeichnet der strukturelle Softwaretest den Prozess, einen in der Implementierung inhärent vorhandenen Umsetzungsfehler aufzudecken, um diesen anschließend beheben zu können. Typische Fehler dieser Art sind Datentypkonvertierungsfehler, negative Logik bei Entscheidungsknoten oder syntaktisch gültige, aber logisch fehlerhafte Ausdrücke [BP84].

2.3.1 Grundlagen des Testens

Ein wesentlicher Erfolgsfaktor des Testens besteht in der Systematisierung des Testprozesses. Dieser lässt sich grundsätzlich in die drei Phasen Vorbereitung, Durchführung und Auswertung unterteilen. In der Vorbereitungsphase müssen die Testfälle gemäß dem intendierten Testziel definiert werden. Ein Testfall besteht aus einer Menge von Einzelschritten über die das zu testende Objekt stimuliert wird. Das zu einem Testfall erwartete Verhalten des Testobjekts (Soll) muss durch ein sog. Testorakel festgelegt werden. Während der Phase der Testdurchführung werden die Testfälle auf das Testobjekt angewendet und das beobachtete Verhalten mit dem Sollverhalten verglichen. Ein Testobjekt kann innerhalb eines Testfalls durch Variation der Stimulationswerte (Testdaten) intensiver getestet werden, d.h. die Testabdeckung und Testtiefe werden dadurch erhöht. Die Ermittlung der geeigneten Menge und konkreten Werte der

Testdaten stellt eine zentrale Herausforderung im Testprozess dar. Das Ergebnis des Soll-Ist-Vergleichs eines Testfalls wird mit einem Verdikt (Ergebnis des Testfalls) in der Auswertephase bewertet. Die Resultate aller Testfälle werden in einem Testreport dokumentiert. Anschließend erfolgt eine Bewertung der Testergebnisse, insbesondere der fehlgeschlagenen Schritte, indem das beobachtete Verhalten analysiert und die dazu notwendige Fehlerbehebung priorisiert wird.

Es müssen stets alle Testfälle abgearbeitet und deren Resultate bewertet werden, bevor Fehler aus dem zu testenden System beseitigt werden. Somit ist sichergestellt, dass Tests immer gegen ein definiertes System einer bestimmten Konfiguration und Version durchgeführt werden. Nach der Fehlerbehebung kann ein weiterer Testlauf durchgeführt werden.

Im Folgenden werden die wesentlichen Grundbegriffe des Testens aufgeführt und deren Bedeutung nach dem ISTQB Glossar beschrieben.

- **Testziel** „Ein Grund oder Zweck für den Entwurf und die Ausführung von Tests.“ [HL14]
- **Testen** „Der Prozess, der aus allen Aktivitäten des Lebenszyklus besteht (sowohl statisch als auch dynamisch), die sich mit der Planung, Vorbereitung und Bewertung eines Softwareprodukts und dazugehöriger Arbeitsergebnisse befassen. Ziel des Prozesses ist sicherzustellen, dass diese allen festgelegten Anforderungen genügen, dass sie ihren Zweck erfüllen, und etwaige Fehlerzustände zu finden.“ [HL14]
- **Testfall** „Umfasst folgende Angaben: die für die Ausführung notwendigen Vorbedingungen, die Menge der Eingabewerte (ein Eingabewert je Parameter des Testobjekts), die Menge der vorausgesagten Ergebnisse, sowie die erwarteten Nachbedingungen. Testfälle werden entwickelt im Hinblick auf ein bestimmtes Ziel bzw. auf eine Testbedingung, wie z.B. einen bestimmten Programmpfad auszuführen oder die Übereinstimmung mit spezifischen Anforderungen zu prüfen (wie Eingaben an das Testobjekt zu übergeben und Sollwerte abzulesen sind). [Nach IEEE 610]“ [HL14]
- **Testdaten** „Daten die (z.B. in einer Datenbank) vor der Ausführung eines Tests existieren, und die die Ausführung der Komponente bzw. des Systems im Test beeinflussen bzw. dadurch beeinflusst werden.“ [HL14]
- **Testeingabe** „Die Daten, die das Testobjekt während der Testdurchführung von einer externen Quelle empfängt. Die externe Quelle kann Hardware, Software oder ein Mensch sein.“ [HL14]
- **Testorakel** „Informationsquelle zur Ermittlung der jeweiligen vorausgesagten Ergebnisse, die mit den tatsächlichen Ergebnissen einer Software im Test zu vergleichen sind.“ [HL14]
- **Testobjekt** „Die Komponente oder das System, welches getestet wird.“ [HL14]
- **Testreport/Testprotokoll** „Eine chronologische Aufzeichnung von Einzelheiten der Testausführung. [IEEE 829]“ [HL14]

2.3.2 Anforderungen an den fehlerorientierten Strukturtest

Im Vergleich zu den allgemeinen Implementierungsfehlern bezieht sich der fehlerorientierte Strukturtest nicht auf die oben genannten Fälle, sondern auf die Überprüfung des Quelltextes der Steuerungssoftware und auf die vorhandene Berücksichtigung ungewollter (fehlerhafter) Maschinenzustände, bspw. Reaktion im Fehlerfall. Aus den Ergebnissen der Umfrage in Abschnitt 2.2.1 ist es ersichtlich, dass exakt jene Testfälle (bzw. Testdaten) identifiziert werden müssen, um Fehlerszenarien stimulieren und das damit verbundene Verhalten beobachten zu können. Aufgrund der hohen Komplexität des Systems und des Parameterraums entspricht dieser Vorgang letztendlich einem Prozess des Ratens [Leh04]. Da der Softwaretest i.d.R. nicht explizit als eigenständige Phase in der Entwicklung einer Maschine oder Anlage berücksichtigt, sondern als integraler Bestandteil der Softwarekonstruktion betrachtet wird, muss der Prozess des Ratens zur objektiven Bewertung systematisiert und zur Effizienzsteigerung automatisiert werden. Die Bedeutung der statischen Analyse des Quelltextes als Qualitätskontrollmechanismus wird besonders in der Sicherheitsnorm IEC 61508 hervorgehoben [IEC02].

Testfall-, respektive Testdatengenerierungsverfahren besitzen meist die Eigenschaft, eine zu große Menge an Testfällen, im Sinne der praktischen Durchführbarkeit zu generieren [KVH11]. Es bedarf deshalb eines Verfahrens, das eine Reduktion auf die relevanten Testdaten ermöglicht, so dass eine größtmögliche Testabdeckung erreicht wird. Des Weiteren ist die Aussagekraft der erzeugten Testfälle stark vom gewählten Kriterium abhängig, welches dem Generierungsansatz zugrunde gelegt wurde abhängig [Mye79]. Die Beschränkung auf Fehlersituationen in der Maschine oder Anlage stellt das Kriterium zur Erhöhung der Aussagekraft bei der Testdatengenerierung dar. In Kombination mit einer geeigneten Fehlermodellierung und der dazugehörigen Spezifikationsgranularität kann über den Bezug zum Fehler eine Skalierung der Testtiefe erreicht werden.

2.3.3 Testdatengenerierung für zyklische Softwarebausteine

Die Stimulation ereignisbasierter Softwarebausteine erfolgt i.d.R. durch einen einmaligen Vorgang[1], d.h. die Quelltextfragmente werden einmal entsprechend dem gegebenen Kontrollfluss abgearbeitet. Eine Generierung der Testeingabedaten auf Basis der Analyse des Quelltextes kann demnach auf einer Auswertung des Kontrollflusses basieren.

1 Auf komplexere Zusammenhänge beim Test ereignisbasierter Systeme, wie die Reihenfolge und Kombination des Auftretens der jeweiligen Ereignisse, wird hier nicht näher eingegangen und liegt außerhalb des Fokus' dieser Arbeit.

Listing 2.1: Beispiel der Abhängigkeiten zyklischer Programme

```
01: IF NOT arrived AND sAngle <> fromPos THEN
02:   motor := 1;
03: ELSIF arrived AND bottle THEN
04:   IF sAngle <> toPos THEN
05:     motor := -1;
06:   ELSE
07:     motor       := 0;
08:     destination := true;
09:   END_IF
10: ELSE
11:   motor    := 0;
12:   arrived := true;
13: END_IF
14: IF destination AND NOT bottle THEN
15:   finished := true;
16: END_IF
```

Bei zyklisch ausgeführter Software, wie den Steuerungsprogrammen einer SPS, werden Programmbausteine wiederholt hintereinander ausgeführt. Mit jedem Durchlauf kann sich eine interne Zustandsänderung ergeben, die in der darauffolgenden Abarbeitung in einer Pfadänderung resultiert. Dies legt in der Abhängigkeit disjunkter Pfade begründet, so dass eine Traversierung der Bausteine, bspw. auf Basis des Kontrollflussgraphen nicht durch eine einmalige Ausführung vollzogen werden kann.

Tab. 2.6: Beschreibung der Variablen aus Listing 2.1

Variablenart	Bezeichner	Datentyp	Bedeutung
Sensorwert	sAngle	INT	Aktuelle Winkelstellung
	bottle	BOOL	Flaschenindikator
Aktorwert	motor	INT	Drehrichtung Motor
Übergabeparameter	from	INT	Ausgangswinkel
	to	INT	Zielwinkel
Lokale Variable	arrived	BOOL	Hilfsvariable Ausgang
	destination	BOOL	Hilfsvariable Ziel
	finished	BOOL	Hilfsvariable Abschluss

Das Beispiel in Listing 2.1 zeigt die Umsetzung der Transportfunktionalität einer Weiche. Die realisierte Funktion beschreibt eine Rotation zum Winkel *from*, um von dort eine Flasche aufzunehmen und diese anschließend zum Winkel *to* zu drehen. Tabelle 2.6 gibt eine Übersicht der in dem Beispiel verwendeten Variablen. Um nun die Testeingangsdaten zur Stimulation zu bestimmen, so dass die Zeile 5 *motor := -1;* erreicht und ausgeführt wird, müssen alle Pfadbedingungen entlang des Kontrollflusses bis zu der entsprechenden Anweisung (Knoten) wahr werden, siehe gestrichelten Pfad in Abbildung 2.4. Alle lokalen Variablen sind anfangs mit *false* initialisiert, so dass der boolesche Ausdruck in Zeile 3 durch keine geeignete Wahl der Übergabeparameter und Sensorwerte wahr werden kann. Infolgedessen ist es nicht möglich, mit einem

einmaligen Durchlauf Testdaten zu generieren, so dass die Anweisung in Zeile 5 mit den vier Bedingungen 2.1 bis 2.4 erreicht wird.

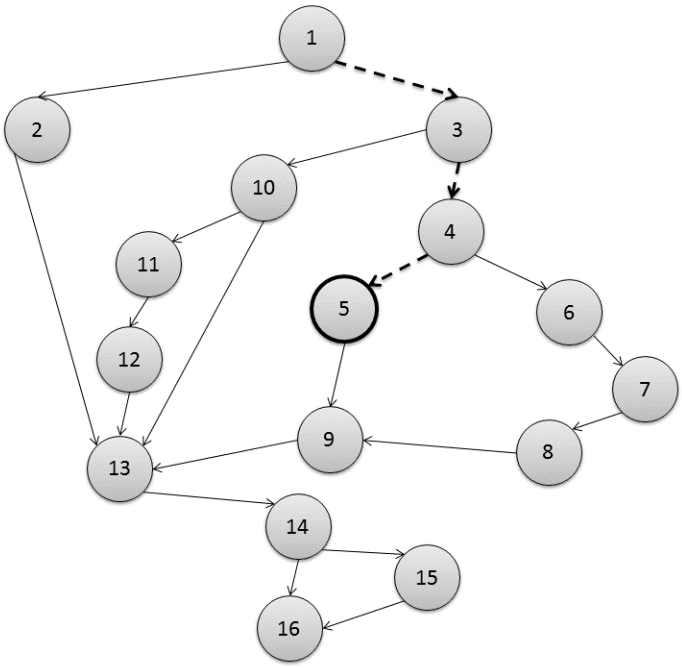


Abb. 2.4: Kontrollfluss des Beispiels in Listing 2.1 mit Pfad zur Zeile 5 (gestrichelt)

$$\text{UNEQUAL}(sAngle, to) \tag{2.1}$$

Der aktuelle Drehwinkel der Weiche *sAngle* darf nicht dem Zielwinkel entsprechen, so dass die Rotation mit dem Ausdruck in Zeile 5 aktiviert wird.

$$\text{EQUAL}(bottle, TRUE) \tag{2.2}$$

Es muss sich eine Flasche in der Weiche befinden, da es sich dabei um das zu transportierende Gut handelt. Nur wenn der Sensor dieses Signal liefert, wird der entsprechende Ausdruck wahr.

$$\text{EQUAL}(arrived, TRUE) \tag{2.3}$$

Die lokale Variable *arrived* muss *TRUE* werden, d.h. die Weiche muss den Ausgangswinkel *from* erreicht haben.

$$\text{NOT}(\text{UNEQUAL}(sAngle, from)) \Leftrightarrow \text{EQUAL}(sAngle, from) \tag{2.4}$$

Da *arrived* zu Beginn mit *false* initialisiert wurde, ist der erste Teil des booleschen Ausdrucks wahr, so dass der zweite Teile falsch werden muss. Dies kann nur dadurch erreicht werden, indem sich die Weiche auf dem Ausgangswinkel *from* befindet.

Es bedarf eines Verfahrens zur Testdatengenerierung für zyklische IEC 61131-3 Softwarebausteine, die die Einschränkungen der Testobjektstimulation aufhebt.

2.3.4 Zusammenfassung

Dieses Kapitel hat die Notwendigkeit und Anforderungen des strukturorientierten Tests aufgezeigt. Eine der zentralen Herausforderungen ist das Ableiten der Testdaten zur Stimulation des Testobjekts. Gängige Verfahren für ereignisbasierte Systeme können, wie gezeigt, nicht unmittelbar auf zyklisch ausgeführte IEC 61131-3 Applikationen angewendet werden, da es in Pfaden unterschiedlicher Subgraphen des Kontrollflusses zu Abhängigkeiten kommen kann und die Pfaderreichung an die mehrfache Ausführung des Testobjekts gekoppelt ist. Ein automatisches Verfahren zur Generierung der Testeingabedaten muss zusätzlich berücksichtigen, dass ein minimaler Satz an repräsentativen Testdaten erzeugt wird, der auf die in Abschnitt 2.2.2 geforderten Fehlermodellen basiert. Die automatische Abarbeitung ist dabei fundamental, da eine manuelle Bestimmung zeitaufwändig und basierend auf den Umfrageergebnissen in Abschnitt 2.2.1 nicht praktikabel ist.

2.4 Handlungsbedarf und Anforderungen

Dieser Abschnitt fasst die Erkenntnisse der Problemanalyse zusammen und erläutert die an eine Lösung zur fehlerorientierten Testdatengenerierung gestellten Forderungen näher.

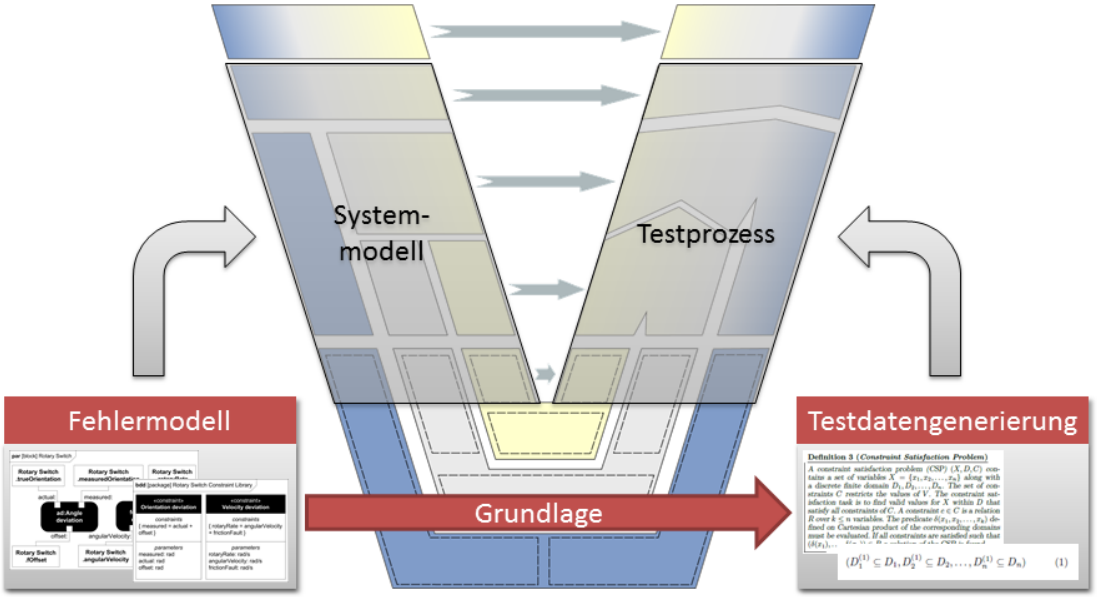


Abb. 2.5: Einordnung der Aufgabenstellung in den Entwicklungsprozess

2.4.1 Aufgabenstellung

Ziel dieser Arbeit ist es, eine Möglichkeit zu schaffen, womit sich zur Absicherung reaktiver Automatisierungssysteme Testdaten zur Stimulation zyklischer Steuerungssoftware für fehlerhafte technische Komponenten (Sensoren, Aktoren) in der Maschine oder Anlage automatisch generieren lassen. Die Grundlage dazu liefert ein Fehlermodell, das in das Systemmodell integriert werden muss und somit die Basis für eine zielgerichtete Generierung der Testdaten darstellt. Die dadurch gewonnenen Testdaten dienen für die im Testprozess zur Ausführung notwendigen Testfälle. Abbildung 2.5 zeigt die Einordnung der Arbeit in den Entwicklungsprozess. Die beiden wesentlichen Aspekte sind die Konzeption und Integration eines Fehlermodells in das Systemmodell (1) und die darauf aufbauende Testdatengenerierung für den Testprozess (2). Das auf Systemebene erstellte Systemmodell dient den nachgelagerten Entwicklungsphasen als Grundlage und kann sowohl disziplinspezifisch als auch -übergreifend detailliert werden. Die Testdatengenerierung stellt in Abhängigkeit der Fehlergranularität einen skalierbaren Mechanismus zur Unterstützung des gesamten Testprozesses.

2.4.2 Anforderungen

Auf Basis des in diesem gesamten Kapitel analysierten Problems ergeben sich zusätzliche Anforderungen, die an eine Lösung der im vorangegangenen Unterkapitel 2.4.1 erarbeiteten Aufgabenstellung gestellt werden. Die Anforderungen an eine Lösung zur fehlerorientierten Testdatengenerierung werden im Folgenden näher erläutert:

- Eine wesentliche Anforderung an die Lösung stellt die Möglichkeit der unterschiedlich granularen Fehlermodellierung dar. Je nach vorhandenem Wissen in den jeweiligen Phasen kann ein Systemfehler anfangs grob beschrieben und bei entsprechender Erfahrung bzw. fortschreitender Entwicklung detaillierter beschrieben werden. Die Modellierungssprache muss die Möglichkeit bieten, Fehler auf unterschiedlichen Detailstufen modellieren zu können.

- Das Fehlermodell muss permanente Fehler von technischen Komponenten der Maschine bzw. Anlage abdecken. Diese können von Verschleiß durch Alterung, d.h. einer Leistungsreduktion bis zum vollständigen Ausfall einer Komponente, reichen.

- Die Generierung der Testdaten auf Basis des IEC 61131-3 Quelltextes speicherprogrammierbarer Steuerungen muss automatisch ablaufen. Aufgrund der hohen zeitlichen Restriktion im Entwicklungsprozess steht für den Testprozess keine explizite Zeit zur Verfügung, so dass alle testrelevanten Aktivitäten mit minimalem Aufwand ablaufen müssen.

- Die generierte Menge an Testdaten muss reduziert sein, d.h. aus jeweils repräsentativen Daten bestehen. Dies wird durch eine geeignete Wahl des Testziels erreicht. Dabei werden einerseits ein Bezug zum Fehler als Aspekt der Einschränkung und andererseits die aus der Strukturanalyse extrahierten, in der Wirkung identischen, Parameterwerte zu Klassen zusammengefasst.

- Der zur Modellierung der Fehler notwendige Aufwand muss gering sein. Dies kann u.a. dadurch erreicht werden, indem die Modellierungssprache an die Bedürfnisse der Domäne Maschinen- und Anlagenbau angepasst ist. Des Weiteren muss die Generierung der Testdaten mit möglichst geringem zusätzlichen Aufwand des Testers verbunden sein.

KAPITEL 3

Modellbildung und Testverfahren im Maschinen- und Anlagenbau

Die Ableitung von aussagekräftigen Testdaten stellt eine zentrale Herausforderung bei der Spezifikation und Durchführung von Tests mechatronischer Systeme dar. Eine Einschränkung auf Fehler im ungesteuerten System wird bislang aufgrund der mangelnden Berücksichtigung während der Systemmodellierung nicht getroffen. Es zeigt sich, dass kein Ansatz zur fehlerzentrierten Modellierung und Testdatengenerierung für IEC 61131-3 basierte Applikationen existiert.

Inhaltsverzeichnis

3.1 Bewertungskriterien

Auf Basis der in Kapitel 2 erarbeiteten Handlungsfelder der Problemanalyse werden die zur Vergleichbarkeit und Bewertbarkeit der Ansätze zur Systemmodellierung und Testverfahren notwendigen Bewertungskriterien eingeführt und im Detail näher erläutert. Diese bilden die Grundlage für die Gesamtbewertung in den Unterkapiteln 3.3.4 und 3.4.5. Der Grad der Erfüllung des jeweiligen Kriteriums wird tabellarisch mit einer dreistufigen Bewertungsskala eingestuft. Die folgenden Grade werden unterschieden:

- ● : Vollständig erfüllt
- ◑ : Teilweise erfüllt / nicht feststellbar
- ○ : Nicht erfüllt

3.1.1 Kriterien zum Vergleich der Systemmodellierungsansätze

Auf Basis der im vorangegangenen Kapitel erhobenen Anforderungen werden im Folgenden sechs relevante Kriterien zur transparenten Bewertung der Systemmodellierungsansätze definiert.

- *Unterstützung von Sichten (**/S1**)* In der modellbasierten Entwicklung ist ein Sichtenkonzept zur gemeinsamen, integrierten Modellbildung von besonderer Bedeutung. Dies unterstützt einerseits die unterschiedlichen Gewerke, ermöglicht eine Reduktion der Komplexität auf die jeweilige Sicht und steigert zugleich die Wiederverwendbarkeit. Dabei werden das technische System, der technische Prozess und das Automatisierungssystem separiert [SW09].
- *Modellierungsgranularität (**/S2**)* Dieses Kriterium umschreibt die Unterstützung der Modellierungssprache, Modelle unterschiedlicher Detailtiefe (Granularität) erstellen zu können. Dies beinhaltet einerseits die hierarchische Dekomposition von gesamten Anlagen bis auf Komponentenebene als auch andererseits die Möglichkeit der iterativen Verfeinerung und dadurch Konkretisierung des integrierten Fehlermodells. Dadurch wird sichergestellt, dass eine Modellbildung während des gesamten Entwicklungsprozesses vorliegt und, an die Entwicklung gekoppelt, durch weitere Detailinformationen erweitert werden kann. Dies unterstützt besonders die Granularität der modellierten Fehler [SLVH09].
- *Graphische Beschreibungssprache (**/S3**)* Die zur Modellbildung von Maschinen und Anlagen und der dazugehörigen Steuerungstechnik notwendige Beschreibungssprache muss eine graphische Notation anbieten. Die Zielgruppe umfasst eine Vielzahl von Personen mit unterschiedlichem Ausbildungshintergrund und akzeptiert vornehmlich graphische Sprachen [HV05], [VHS11].
- *Integration von Fehlern (**/S4**)* Für die zielgerichtete Kriterienauswahl zur Testdatengenerierung wird eine explizite Integration von Fehlern bzw. Fehlermodellen in das Gesamtsystemmodell notwendig. Das Fehlermodell stellt die Grundlage zur Generierung reduzierter und aussagekräftiger Testdaten.

- *Maschinen- und Anlagenbau (**/S5**)* Dieses Kriterium dient zur Klassifikation des Einsatzbereiches der jeweiligen Ansätze, da unterschiedliche Anforderungen in den Branchen Automobil und Luft-/Raumfahrttechnik im Vergleich zum Maschinen- und Anlagenbau vorherrschen [KFVH12], [KTVH12], die u.a. auch in der deutlich geringeren Stückzahl begründet liegen. Daraus ergeben sich hohe Effektivitätsbestrebungen für das gesamte Engineering im Maschinen- und Anlagenbau.

- *IEC 61131-3 (**/S6**)* Dieses Kriterium stellt eine spezielle Ausprägung des vorherigen Kriteriums dar, welches als Differenzierung der Eignung der Modellierungsansätze für zeitschlitzbasierte (zyklische) Systeme im Vergleich zu ereignisbasierten Systemen dient [Thr05], [ZSSB09], [SZC08].

3.1.2 Kriterien zum Vergleich der Testverfahrensansätze

Auf Basis der im vorangegangenen Kapitel erhobenen Anforderungen, werden im Folgenden sieben relevante Kriterien zur transparenten Bewertung der Testverfahren definiert.

- *Fehlerbezogen (**/T1**)* Testverfahren für Software zielen i.a. darauf ab, strukturelle und funktionale Implementierungsfehler aufzudecken. Zur Absicherung reaktiver Systeme ist jedoch das Fehlerverhalten von Komponenten im gesamten technischen System des Maschinen- und Anlagenverbunds von besonderem Interesse. Die Generierung der Testdaten wird demnach daraufhin ausgelegt, dass das Steuerungsprogramm während des Tests auf eine defekte Komponente zugreift [KVH11].

- *Testdatenreduktion (**/T2**)* Automatisch generierte Testfälle besitzen im allgemeinen eine zu breite Datenbasis, die der Testdatengenerierung zugrunde gelegt wird, da keine präzise Selektion des Testziels erfolgt. Testfälle müssen in endlicher Zeit ausgeführt werden können. Deshalb wird eine geringe Auswahl an Testdaten benötigt, die eine möglichst große Fehleraufdeckungsrate besitzen [Dem78], [GG75]. Dieses Kriterium gibt an, ob in dem jeweiligen Ansatz ein derartiger Mechanismus integriert wurde.

- *Ausführungsart (**/T3**)* Bei dem Kriterium der Ausführungsart wird zwischen automatisch, teilautomatisch und manuell unterschieden. Für den Maschinen- und Anlagenbau ist es von besonderer Relevanz, dass die Generierung der testrelevanten Daten vollständig automatisch abläuft, da alle Tätigkeiten während des Engineerings aufwandsarm durchgeführt werden müssen, d.h. eine Generierung von Testdaten sollte keinen wesentlichen zusätzlichen Aufwand darstellen [KTVH12].

- *Zyklische Ausführungslogik (**/T4**)* Testverfahren haben sich besonders im Bereich der Hochsprachenprogrammierung etabliert. Diese Ansätze sind auf Modell- und Codeebene für ereignisbasierte Systeme ausgelegt. IEC 61131-3 basierte Applikationen besitzen jedoch eine zyklische Ausführungslogik zur Steuerung

der internen Logik. Gängige Verfahren berücksichtigen diese Eigenschaft nicht explizit. Diese Betrachtung stellt umfassende Anforderungen an das Generierungsverfahrens und wodurch dieses Kriterium ein zentraler Bestandteil der Bewertung aller Ansätze ist [KVH11].

- *Strukturorientiert (**/T5**)* Verfahren zur Testgenerierung, die auf einer Spezifikation beruhen, bergen die Gefahr, inkonsistente Tests zu erzeugen, da während der Entwicklung technischer Systeme Drifts zwischen der Spezifikation und der Umsetzung auftreten können. Die Generierung der Testdaten, basierend auf der tatsächlichen Umsetzung, ermöglicht eine präzise Aussage über die tatsächlich vorhandene Funktionalität [Mye79]. Dieses Kriterium besagt, ob in den jeweiligen Testverfahren Tests aus einer Spezifikation oder der Quelltextstruktur generiert werden.

- *Geringer Aufwand (**/T6**)* Eine stärkere Ausprägung der Ausführungsart. Hiermit wird der Gesamtaufwand zur Erstellung/Generierung von Testfällen bzw. Testdaten betrachtet. Testverfahren müssen entsprechend allen Aufgaben des Engineerings im Maschinen- und Anlagenbau aufwandsarm durchführbar sein [KTVH12], [KFVH12].

- *Aussagekraft der Testdaten (**/T7**)* Testgenerierungsverfahren zeichnen sich häufig dadurch aus, dass die generierte Menge entweder gegen unendlich geht oder zumindest im Sinne der Durchführbarkeit und somit der praktischen Relevanz zu umfangreich oder falsch gewählt ist. Deshalb ist es fundamental, dass die Aussagekraft der generierten Testdaten auf das zu testende Kriterium hoch ist [Dem78].

3.2 Fehler in mechatronischen Systemen

In diesem Kapitel erfolgt eine Klärung des Fehler-Begriffs. Bei der Entwicklung von technischen Systemen können Fehler drastische Konsequenzen für das Unternehmen (Kostenexplosion, Imageschaden) oder die Umwelt (Gefährdungspotential) nach sich ziehen. Die Auswirkungen lassen sich auf unterschiedliche Entstehungsbereiche zurückführen. So können Fehler entlang des gesamten Entwicklungsprozesses (Makro- und Mikrozyklus), aufgrund einer mangelhaften Spezifikation oder fehlerbehafteten Umsetzung während der Entwicklung und auch während des späteren Betriebs auftreten. Ein typisches Beispiel für den aufgrund eines Softwarefehlers entstandenen Schadens (ca. 1,7 Milliarden DM) ist die Explosion der unbemannten Rakete Ariane 5 während des Jungfernflugs am 04.06.1996. Als Ursache dafür wurde in der Steuerungssoftware ein Überlauf nach der Konvertierung einer 64 Bit Gleitkommazahl in eine vorzeichenbehaftete 16 Bit Ganzzahl ausgemacht. Wenngleich die Software bei dieser Datentypkonvertierung ein korrektes Verhalten aufwies, hatte die damit verbundene Konsequenz den Absturz bzw. die Selbstzerstörung der Rakete zur Folge, so dass die entsprechende Stelle im Quelltext als Softwarefehler bezeichnet wird.

Systeme besitzen für sich betrachtet keine Fehler, da sie sich gemäß den physikalischen Gesetzmäßigkeiten verhalten und insbesondere jenes Verhalten aufweisen,

welches sowohl beabsichtigt als auch unbeabsichtigt implementiert wurde. Ein Fehler ist demnach kein absolutes Maß, sondern muss stets als eine relative Größe, d.h. eine Abweichung in Bezug auf bspw. eine Erwartungshaltung oder eine Spezifikation betrachtet werden. Demnach hat die Einschätzung, ob es sich bei einem beobachteten Systemverhalten um einen Fehler handelt mitunter einen subjektiven Charakter. In der Mathematik wird zwischen dem absoluten F_a (Gleichung 3.1) und dem relativen F_r (Gleichung 3.2) Fehler unterschieden. Dabei wird jeweils der Messwert x_m mit dem tatsächlichen (erwarteten) Wert x_r in Relation gesetzt, so dass die Abweichung als Fehler interpretiert wird.

$$F_a = x_m - x_r \tag{3.1}$$

Fehler in mechatronischen Systemen können sowohl bekannt als auch unbekannt sein. Während des Betriebs aufgedeckte Fehler werden nach der Analyse auf ihre Eintrittswahrscheinlichkeit und Kritikalität hin bewertet. Anschließend erfolgt üblicherweise eine Entscheidung der Notwendigkeit der Fehlerbehebung. Durch die Behebung der bekannt gewordenen Fehler wird das entwickelte System iterativ optimiert. Unbekannte Fehler sind jene, die während der Entwicklung und anschließenden Verifikation nicht aufgedeckt werden konnten und sind deshalb besonders kritisch, da deren mögliche Auswirkung auf das System und die Umwelt nicht abgeschätzt werden kann. Deshalb ist es das Ziel, die Anzahl der in einem System vorhandenen Fehler (relativ der jeweiligen Vorgabe) bereits während der Entwicklung aufzudecken und zu beheben.

$$F_r = \frac{x_m - x_r}{x_r} \tag{3.2}$$

Im Folgenden werden einige Definitionen von Fehlern näher erläutert. Definition 2 bezeichnet einen Fehler als die Nichterfüllung einer Anforderung. Da die Qualität i.d.R. als der Grad der Anforderungserfüllung bezeichnet wird, entspricht diese Definition dem weitläufigen Verständnis für einen Fehler. Dabei bleibt ungeklärt, ob es sich bei den genannten Anforderungen um jene handelt, die während der Anforderungserhebungs- und detaillierungsphase zwar erhoben, jedoch nur unzureichend oder gar nicht umgesetzt wurden oder ob es auch alle Anforderungen miteinbezieht, die gänzlich unberücksichtigt (vergessen) wurden. Sowohl Definition 3 nach einer Instandhaltungsnorm als auch Definition 4 nach einer Norm für funktionale Sicherheit wird ein Fehler als ein Zustand bzw. eine Bedingung bezeichnet, die ein System einnimmt und zugleich eine geforderte Funktion nicht mehr vollständig erfüllen kann. Dabei werden sowohl die jeweilige Ursache (bspw. vergessene Anforderung) für die Manifestierung des Fehlers sowie der Grad der möglichen Konsequenzen beim Eintreten eines Fehlers nicht näher spezifiziert.

Definition 2 (*DIN EN ISO 9000, [DIN05]*)

> *Nichterfüllung einer Anforderung.*

Definition 3 (*DIN EN 13306, [DIN10]*)

Zustand einer Einheit, in dem sie unfähig ist, eine geforderte Funktion zu erfüllen; ausgenommen die Unfähigkeit während der präventiven Instandhaltung oder anderer geplanter Maßnahmen oder infolge des Fehlens externer Hilfsmittel.

Definition 4 (*DIN EN 61508, [DIN02]*)

Nicht normale Bedingung, die eine Verminderung oder den Verlust der Fähigkeit einer Funktionseinheit verursachen kann, eine geforderte Funktion auszuführen. Bezüglich der Fehlerursache sind dabei zwei typische Fehlerkategorien, und zwar zufällige Fehler und systematische Fehler zu unterscheiden. Hinsichtlich ihres Erscheinungsbildes ist weiterhin zwischen intermittierenden Fehlern [Transient Faults] und permanenten Fehlern [Solid Faults] zu unterscheiden.

3.2.1 Fehler-Terminologie

Bei der Definition von Fehlern kommt es häufig zu Mehrdeutigkeiten, da die deutsche Sprache eine nicht eindeutige, präzise Unterscheidung der Fehler Begrifflichkeiten vorsieht. In der reinen Softwareentwicklung werden gängigerweise die folgenden zwei Fehlerkategorien unterschieden:

- **Design-Time Error** sind Fehler, die zur Entwurfszeit entstehen. Häufig handelt es sich hierbei um syntaktische Fehler in der eingesetzten Beschreibungssprache, wie dem Quelltext einer Programmiersprache (bspw. C++) oder dem Aufbau einer Oberflächenbeschreibungssprache (bspw. XAML). Diese Fehlerart ist i.d.R. durch den Einsatz existierender Werkzeuge, wie Compiler bzw. Parser relativ einfach aufzudecken.

- **Runtime Error** beschreiben jene Fehler, die zur Laufzeit auftreten. Dabei kann es sich sowohl um organisatorische Fehler (late binding), aber auch um logische Fehler bzw. unvollständig umgesetzte Funktionalität handeln. Diese Fehlerart ist meist nicht trivial zu identifizieren und zu beheben. Dazu müssen sowohl Werkzeuge als auch methodische Herangehensweise über alle Teststufen (Komponententest, Integrationstest, Systemtest, Abnahmetest) [Cle10] angewendet werden, um die unterschiedlichen Laufzeitfehler aufdecken zu können.

Neben dieser Fehlerunterscheidung lassen sich Fehlertypen nach [RS03] in einen kausalen Zusammenhang bringen, wie in Abbildung 3.1 dargestellt. Dabei erfolgt eine Unterscheidung nach Fehlerursache (fault), Fehler (error) und Funktionsausfall (failure). Die Definitionen aus dem Kapitel 3.2 beschreiben dabei im Wesentlichen den Typ Fehler. Die Quelle eines jeden Fehlers bildet dabei die Fehlerursache und sofern ein System den Fehlerzustand tatsächlich einnimmt, besteht die anschließende potentielle Möglichkeit des Funktionsausfalls.

Zur näheren Erläuterung der Fehlertypen dient im Folgenden der Softwarebug im Stellwerk des Bahnhofs Hamburg Altona aus dem Jahr 1995 [Ber03]. Aufgrund des hohen Arbeitsaufkommens der Mitarbeiter zur Steuerung der 160 Weichen wurde 1995 ein elektronisches Stellwerk in Betrieb genommen. Die Kosten des einjährigen Projektes

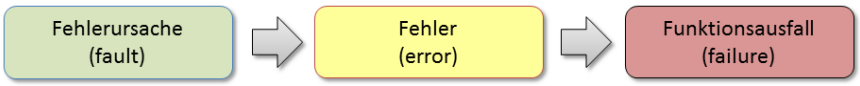


Abb. 3.1: Kausaler Zusammenhang der Fehlertypen [RS03]

beliefen sich auf ca. 60 Millionen DM. Nach der Inbetriebnahme des Systems folgte eine automatische Sicherheitsabschaltung. Dabei fuhr das komplette System nach wenigen Stunden wieder herunter und brachte den gesamten Schienenverkehr dadurch zum Stillstand. Die Abbildung 3.2 zeigt den kausalen Zusammenhang der aufgetretenen Fehler, deren Ursachen und den anschließenden Funktionsausfall nach der Analyse. Die ursprüngliche Fehlerursache lässt sich auf einen zu klein proportionierten Stack zurückführen, der sich bereits beim regulären Schienenverkehr zum Fehler manifestierte und überlief. Dies führte nicht unmittelbar zu einem Funktionsausfall, da für derartige Situationen eine Fehlerbehandlungsroutine vorgesehen war. Diese war jedoch fehlerhaft implementiert, so dass dies die Ursache für eine Endlosschleife im System darstellte. Weitere Sicherheitsmechanismen lösten deshalb eine Sicherheitsabschaltung aus, so dass es zum vollständigen Dienstausfall kam.

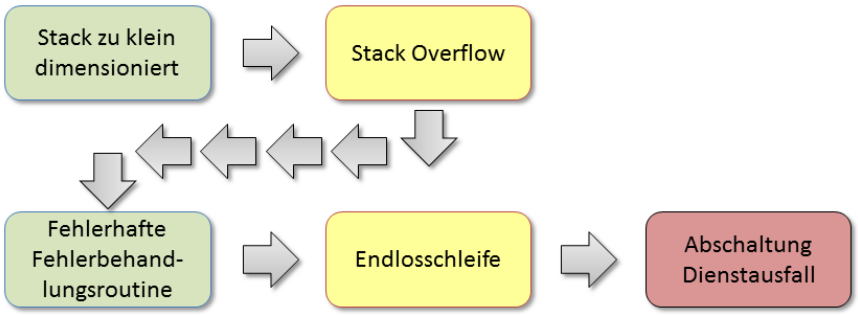


Abb. 3.2: Kausaler Zusammenhang der Fehler des Zentralrechners am Bahnhof Hamburg Altona

3.2.2 Klassifikation von Fehlern und Fehlerursachen

3.2.2.1 Klassifikation von Fehlerursachen

Die im vorherigen Kapitel eingeführte Terminologie zur präzisen Bezeichnung von Fehlern dient als Grundlage für die Klassifikation von Fehlerursachen und Fehlern. Fehlerursachen lassen sich in sechs wesentliche Merkmale kategorisieren, siehe Abbildung 3.3 und können sich in ihrer Form als systematisch (bspw. Drifts) oder zufällig (bspw. thermisches Rauschen) unterscheiden. Zusätzlich lassen sie sich nach ihrer Art als gutmütig (benign) oder bösartig (malicious) einstufen. Eine gutmütige Fehlerursache ist im Regelfall bekannt und pflanzt sich nicht derart fort, so dass das Systemverhalten zu einem Dienstausfall führt. Ein typisches Beispiel dafür ist eine verzögerte Ausführung einer Systemfunktionalität (Timing). Bösartige Fehlerursachen sind nicht offensichtlich identifizierbar und werden auch als byzantinische Fehler bezeichnet. Diese können sowohl symmetrisch (identisches Auftreten) als auch asymmetrisch (beliebiges Auftreten) in Erscheinung treten [LSP82], [MP91], [TP88]. Von besonderer Bedeutung ist die

Dauer einer Fehlerursache. Permanente Fehler können zu einem vollständigen Ausfall führen und sind i.d.R. aufgrund der fortdauernden Existenz im Vergleich zu temporär auftretenden Fehlern reproduzierbar und können somit zielgerichtet diagnostiziert werden. Des Weiteren können Fehler während der Laufzeit als auch in der Entwurfsphase in das System injiziert werden und können dabei einen physikalischen (bspw. Temperaturschwankungen, Messungenauigkeiten etc.) oder menschlichen Ursprung aus Sicht der Ausbildung [VHBO11] besitzen.

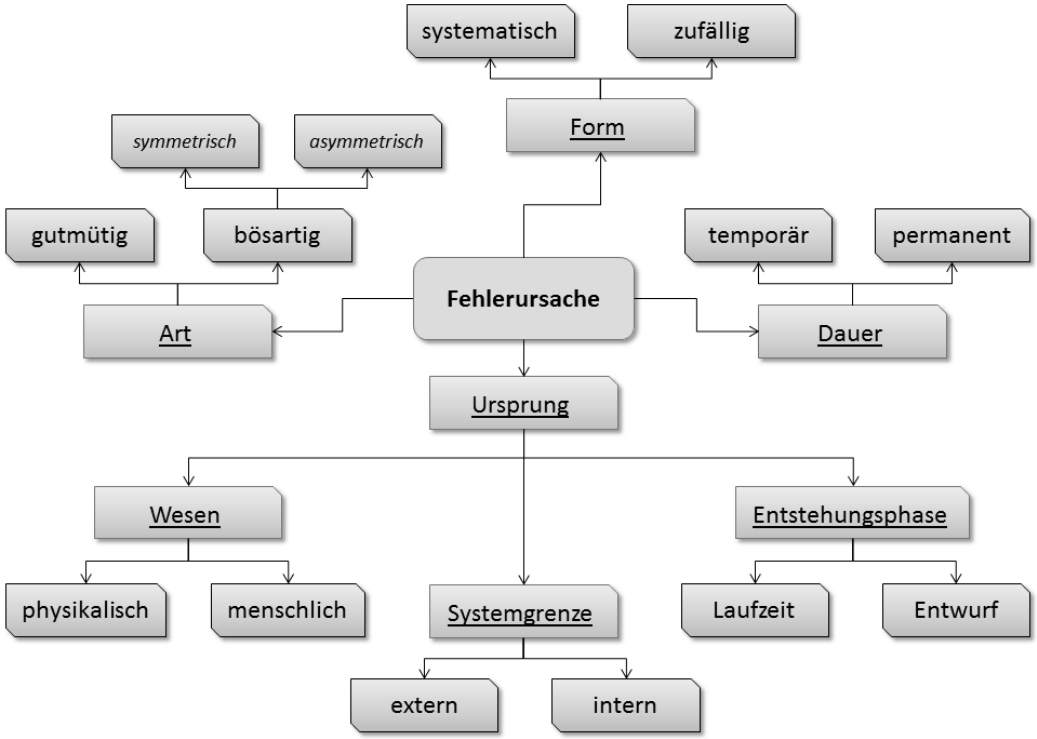


Abb. 3.3: Taxonomie von Fehlerursachen basierend auf [Sch07], [GRSV06], [Muk08]

3.2.2.2 Einteilung in Fehlerklassen

Die Taxonomie der Fehlerursachen wird in sechs Ursprungskategorien eingeteilt. Jeder individuelle Fehler kann darüber eindeutig eingestuft werden. Es kann jedoch kein allgemeiner Zusammenhang zwischen der Ursache und den möglichen Folgen eines Fehlers hergestellt werden. Die DIN 55350-31 stuft Fehler in Fehlerklassen gemessen an ihren entsprechenden Fehlerfolgen ein, siehe Abbildung 3.4. Die Fehlerklassen kritischer Fehler, Hauptfehler und Nebenfehler sind in der Norm gemäß den Definitionen 5 bis 7 definiert.

Definition 5 (*Kritischer Fehler, [DIN85]*)

Fehler, von dem anzunehmen [..] ist, daß er voraussichtlich für Personen [..] gefährliche oder unsichere Situationen schafft oder [..] die Erfüllung der Funktion einer größeren Anlage [..] verhindert.

Definition 6 (*Hauptfehler, [DIN85]*)

Nicht kritischer Fehler, der [..] die Brauchbarkeit für den vorgesehen Verwendungszweck wesentlich herabsetzt.

Definition 7 (*Nebenfehler, [DIN85]*)

Fehler, der voraussichtlich die Brauchbarkeit [..] nicht wesentlich herabsetzt oder ein Abweichen von den geltenden Festlegungen, das den Gebrauch [..] nur geringfügig beeinflusst.

Abb. 3.4: Einteilung von Fehlern in Klassen basierend auf möglichen Folgen [DIN85]

3.2.2.3 Fehlerarten in mechatronischen Systemen

Je nach Art der Maschine oder Anlage wirkt ein Alterungsprozess auf die eingesetzten Komponenten, der stark durch die äußeren Einflussfaktoren der Umwelt geprägt ist. Zu den zentralen Faktoren zählen hierbei Feuchtigkeit und Temperaturvariationen. Dies zieht zumeist einen Verschleiß der Mechanik nach sich, was wiederum eine veränderte Reibung oder Sensordrifts hervorgerufen und ein Fehlverhalten zur Folge haben kann. Die Ausfallrate mechanischer Baugruppen verläuft nach [FB04] und [Bor08] entsprechend der Badewannenkurve[1]. In den frühen Phasen erfolgen häufig Konstruktions-, Montage- und Produktfehler, die zufälligen Fehler umfassen Bedienfehler oder Verschmutzungen und nach einer langen Laufzeit führt der erhöhte Verschleiß zu einer Funktionsänderung [Bor08].

In mechatronischen Systemen können Fehler in jeder beteiligten Disziplin (Mechanik, E/E, Software) auftreten. Tabelle 3.1 zeigt eine Übersicht über gängige Fehlerursachen in mechatronischen Systemen nach [BGJ09], [Ise07]. Die Klasse der zufälligen Fehler gilt als besonders kritisch, da sie i.d.R. schwer diagnostizierbar sind. Ein zufälliger Fehler zeichnet sich nach DIN 1319-1 [DIN95] dadurch aus, dass er eine „Abweichung des

1 Im ersten Abschnitt (Frühausfälle) treten hohe, aber abnehmende Fehlerraten auf. Im mittleren Abschnitt treten i.d.R. zufällige Ausfälle auf, die eine konstante Ausfallrate aufweisen. Im letzten Abschnitt erhöht sich die Fehlerrate wieder, was auf die zunehmende Alterung der Komponenten zurückzuführen ist.

unberichtigten Meßergebnisses vom Erwartungswert“ darstellt. Als mögliche Ursachen werden genannt:

- „nicht beherrschbare Einflüsse der Meßgeräte“
- „nicht beherrschbare Änderungen der Werte der Einflußgrößen“
- „nicht beherrschbare Änderungen des Wertes der Meßgröße“
- „nicht einseitig gerichtete Einflüsse des Beobachters“

Tab. 3.1: Erweiterte Übersicht der üblichen Fehlerursachen in mechatronischen Systemen basierend auf [BGJ09], [Ise07]

Disziplin	**Fehlerursache**	**Eigenschaften/Details**
Software	systematisch	Spezifikation, Implementierung
Elektronisch	systematisch	Spezifikation, Entwurf
	zufällig	Betrieb
Elektrisch	zufällig	Kurzschluss, Kabelbruch, Kontaktproblem, Korrosion, EMV-Störung
Mechanisch	zufällig	Schlagartig: Überlastung, Ermüdung
	systematisch	Driftartig: Abnutzung, Korrosion

In dem Artikel *Diagnosesysteme für Feldgeräte* von Schneider [Sch99] werden die möglichen Fehler der Feldgeräte in Geräte- und Geräteumgebungsfehler unterteilt, siehe Abbildung 3.5. Von besonderer Relevanz sind die Geräteumgebungsfehler, da diese aus der Interaktion mit der Umwelt resultieren. Dabei wird deutlich, dass analog zu [Bor08] Abweichungen zu einem Fehlverhalten des gesamten Systems führen können.

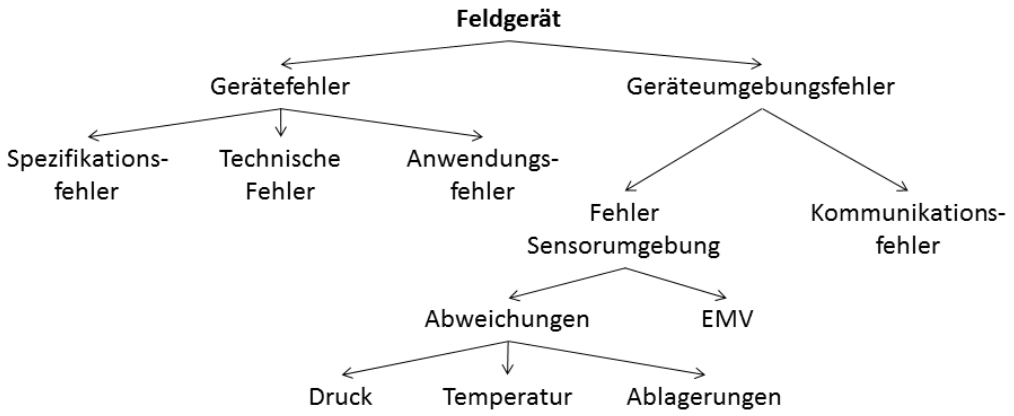


Abb. 3.5: Unterteilung von Feldgerätefehlern nach [Sch99]

3.3 Modellierung technischer Systeme

Technische Systeme sind eine Kompositionen zahlreicher Komponenten und erfüllen gemeinsam eine oder mehrere Funktionen. Häufig setzen sich technische Systeme aus mehreren physikalischen Teildisziplinen wie Mechanik, Hydraulik, Pneumatik uvm. zusammen, die in Kombination mit elektrotechnischen Schaltungen und Softwarebausteinen an Komplexität zunehmen. Zur Beherrschung der Systemkomplexität und konsistenten Entwicklungen haben sich in der Softwareentwicklung modellbasierte Verfahren etabliert [SV07].

3.3.1 Graphische Modellierungssprachen

3.3.1.1 Die Unified Modeling Language (UML)

Die UML (aktuelle Version 2.4.1) stellt einen etablierten Standard für die Softwareentwicklung dar, der durch die Object Management Group (OMG) spezifiziert wird [Gro10a], [Gro10b]. Sie stellt eine graphische Spezifikationssprache zur Visualisierung, Konstruktion und Dokumentation von Entwicklungsartefakten objektorientierter Systeme dar. Die UML kann als Verständnismodell in Form von skizzenhaften Beschreibungen und als ausführbares Gesamtmodell eingesetzt werden. Die UML bietet einen Erweiterungsmechanismus zur Anpassung der Sprachen an domänenspezifische Bedürfnisse. Die Meta Object Facility (MOF) der OMG beschreibt eine Referenzarchitektur zur ebenenbasierten Modellierung von konkret (M0) bis zum abstrakten Meta-Metamodell (M3) [OMG06]. Eine Erweiterung der Sprachfamilie UML wird dadurch realisiert, dass die Sprachmittel der darüber liegenden Ebene zur Beschreibung der Erweiterung verwendet werden. Somit wird das Metamodell (M2) mit den Sprachmitteln der Ebene M3 erweitert. Eine zusätzliche Erweiterungs- bzw. Anpassungsmöglichkeit ist durch die sog. UML-Profile vorgesehen, die den Sprachwortschatz weiter einschränken. Sie bestehen aus Stereotypen, Tagged Values und Constraints. Das UML Testing Profile erweitert die UML um relevante Aspekte des modellbasierten Tests und füllt dadurch die Lücke zwischen Entwurf und Test [OMG12]. Das Profil bietet einen konzeptionellen Rahmen für die Testarchitektur, die Testdaten, das Testverhalten und die Testzeiten. Durch die Wiederverwendung der Entwurfsentitäten wird die Testentwicklung und -durchführung in den frühen Phasen gefördert. In [BDG04] wird das UML Testing Profile am Beispiel eines Bankautomaten erläutert.

Im Maschinen- und Anlagenbau existieren zahlreiche Ansätze zur Integration der UML in die SPS Steuerungsprogrammierung. Zur strukturellen Beschreibung (Modulbildung, Hierarchisierung) einer Steuerungssoftware bietet das UML Klassendiagramm alle notwendigen Beschreibungselemente [WSVH08]. Für die Beschreibung des Verhaltens eignen sich sowohl das Zustandsdiagramm als auch das Aktivitätsdiagramm. Das Zustandsdiagramm bietet eine direkte und eine der Funktionsweise einer Maschine analoge Beschreibungs- bzw. Spezifikationsform. Für die Integration in eine Entwicklungsumgebung und zur Ausführung von in Zustandsdiagrammen programmierten Applikationen ist eine Adaption der Semantik an die zyklische Ausführungslogik der SPS notwendig [WRK10]. Das UML Aktivitätsdiagramm ist eine für Prozessingenieure intuitive Beschreibungssprache von Anlagenprozessen, da sie den objektübergeordneten

Ablauf beschreibt und insbesondere für die Beschreibung von verfahrenstechnischen Prozessen geeignet ist [BVH10]. In [VHBO12] werden Experimente beschrieben, die die Anwendbarkeit von ausgewählten UML Diagrammen im Vergleich zur IEC Sprache FUP für die SPS Programmierung durch Probandenversuche evaluieren. Dabei stellte sich heraus, dass die UML Programmierung keine Nachteile besitzt. Neben der Programmierung von Steuerungsprogrammen stellt auch die Verifikation von SPS Programmen eine weitestgehend ungelöste Aufgabe dar [KFVH12]. Die UML Sequenzdiagramme bieten die Möglichkeit der graphischen Beschreibung von konkreten Ablaufschritten. In [KTVH12] wird ein Ansatz zur Beschreibung von Testfällen für SPS Applikationen mit Hilfe von UML Sequenzdiagrammen präsentiert. Dazu wurde die Semantik der Diagramme an die zyklische Ausführungslogik der SPS angepasst, so dass eine zur Ausführung notwendige automatische Generierung des Testcodes möglich ist.

Die UML ist primär dafür geeignet, softwareintensive Systeme zu beschreiben. Für die Modellierung mechatronischer Systeme, die je nach Domäne disziplinbezogene Aspekte berücksichtigen muss, wurde die SysML als Erweiterung der UML, durch die OMG entwickelt [OMG10].

3.3.1.2 Die Systems Modeling Language (SysML)

Die Systems Modeling Language (SysML) ist eine Sprache zur Beschreibung der Phase des Systems-Engineerings. Die OMG erstellt und verwaltet die Spezifikation der Sprache. Die aktuelle Version 1.2 wurde am 02.06.2010 veröffentlicht [OMG10]. Die SysML basiert konzeptionell auf der UML Superstructure Spezifikation [Gro10b], erweitert diese gemäß den standardisierten Erweiterungsmechanismen und stellt somit eine einheitliche Modellierungssprache dar, die aufgrund des gemeinsamen Metamodells von UML und SysML die modellbasierte Zusammenarbeit zwischen System- und Softwareexperten ermöglicht [OMG10, Seite 1]. Mit Hilfe der zur Verfügung stehenden Beschreibungselemente können alle Aspekte der Anforderungserhebung, Spezifikation, Analyse, Design, Verifikation und Dokumentation von komplexen Systemen bestehend aus Mechanik, Hardware, Software, Prozesse etc. einheitlich erfasst und dargestellt werden. Dadurch wird die durchgängige Modellierung von Systemen über den gesamten Entwicklungsprozess von den Anforderungen bis zum Abnahmetest unterstützt.

Zur Modellbildung werden in der aktuellen Version der Spezifikation neun Diagrammtypen definiert, deren konkrete Syntax aus graphischen Elementen besteht. Die Taxonomie der SysML Diagramme unterscheidet im wesentlichen Struktur- und Verhaltensmodelle der zu beschreibenden Systeme.

Eine zentrale Herausforderung bei der Entwicklung komplexer Systeme ist die Nachverfolgbarkeit und Abhängigkeit von Anforderungen zu Modellelementen. Diese Kenntnis wird für die Bewertung der Testabdeckung und der Testtiefe sowie für effektives Change Management benötigt. Dadurch wird die Erstellung von Qualitätsnachweisen über den gesamten Entwicklungsprozess unterstützt. Bei Änderungen von Kundenanforderungen sind die Auswirkungen auf das bestehende System für die Aufwandsabschätzung der Entwicklung von zentraler Bedeutung. Durch die Tracing Unterstützung über alle Modellelemente der SysML können auch Anforderungsänderungen bis auf die unterste Modellebene nachvollzogen werden.

Taxonomie der SysML Diagramme: Für die Systemmodellierung werden in der SysML Struktur- und Verhaltensmodelle unterschieden. Abbildung 3.6 zeigt die Taxonomie der SysML Diagramme. In dieser Übersicht wird der fundamentale Bezug zur UML durch die gemeinsame Diagrammbasis deutlich. Sieben der neun Diagramme sind vollständig übernommen oder wurden entsprechend für die Systemmodellierung angepasst. Eine wesentliche Neuerung und für die Durchgängigkeit von zentraler Bedeutung nimmt das Anforderungsdiagramm (englisch: Requirement Diagram) ein, das eine strukturierte Anforderungserhebung und -klassifikation ermöglicht.

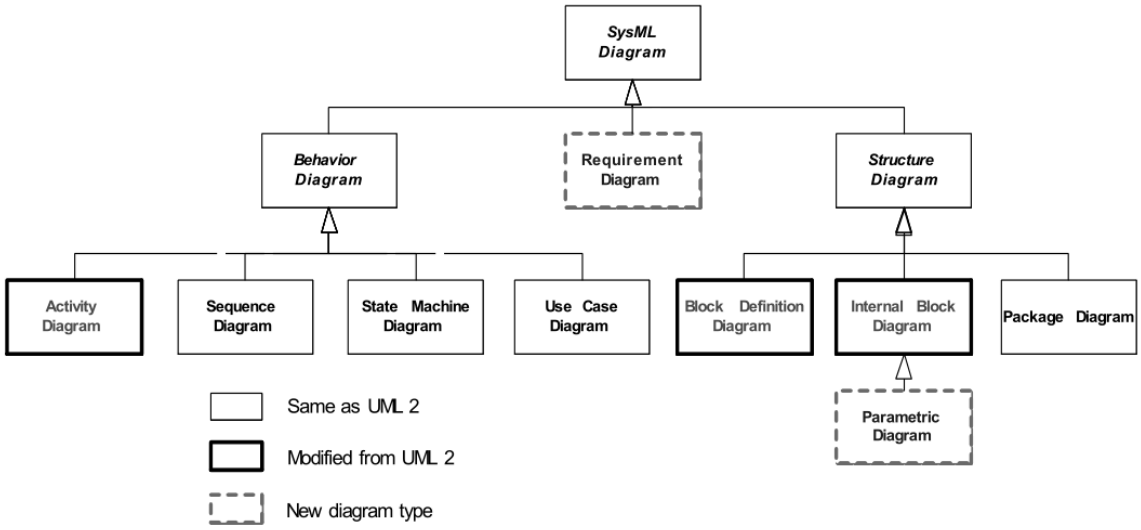


Abb. 3.6: Übersicht der SysML Diagramme [OMG10]

Strukturdiagramme: Zur Beschreibung des strukturellen Modells bietet die SysML vier Diagramme. Das zentrale Diagramm stellt dabei das *Blockdefinitionsdiagramm (BDD)* dar, womit der strukturelle Aufbau von Systemkomponenten und deren hierarchische Zusammenhänge beschrieben werden können. Dieses Diagramm repräsentiert die Black Box Sicht auf das zu modellierende System und findet seine Anwendung bei der Erstellung des Grobentwurfs. Als grundlegendes Element werden sog. Blöcke eingesetzt, die auf dem Klassenkonzept der UML basieren und um Einschränkungen und Erweiterungen in der SysML definiert werden [OMG10]. Informations-, Materie- sowie Energieflüsse zwischen Blöcken werden über Ports modelliert. Das *interne Blockdiagramm (IBD)* basiert auf dem Kompositionsstrukturdiagramm der UML und stellt die White Box Ansicht dar und wird für den Feinentwurf eingesetzt. Darüber werden die internen Zusammenhänge und die verwendeten Schnittstellen (physikalisch, Daten etc.) typenbezogen (Art des Austauschs, bspw. Kraft) modelliert. Die SysML besitzt im Vergleich zur UML das Parameterdiagramm (PAR) zur Notation von mathematischen Zusammenhängen in Form von Constraints (Zusicherungen) zwischen Parametern eines Blocks. Neben der Beziehung zwischen Blöcken, ist das Konzept der Zusicherungen auch auf Zeiteigenschaften und Systemzustandsänderungen (Änderung des Aggregatzustandes in Abhängigkeit der Temperatur) anwendbar.

Verhaltensdiagramme: Die SysML bietet insgesamt vier Diagrammarten zur Verhaltensbeschreibung von Systemen. Das *Zustandsdiagramm* ist für die zustandsbasierte Verhaltensbeschreibung von Komponenten geeignet, da es eine direkte und zur Spezifikation von Komponenten eines technischen Systems intuitive Form bietet. Im Vergleich zu anderen Beschreibungsformen besitzt das Zustandsdiagramm durch die Mächtigkeit der Sprache eine hohe Ausdrucksfähigkeit (u.a. Parallelität, Ausnahmebehandlung, Entscheidungsknoten) [WVH11]. Das Aktivitätsdiagramm beschreibt Kontroll- und Objektflüsse zwischen Aktions-, Objekt- und Kontrollknoten. Im Vergleich zum Zustandsdiagramm werden Aktivitätsdiagramme zur objektübergreifenden Ablaufbeschreibung verwendet und bieten neben den kontrollflussrelevanten Ausdruckselementen auch die Möglichkeit, den Datenaustausch zwischen den Komponenten zu modellieren. Im Maschinen- und Anlagenbau finden Aktivitätsdiagramme Anwendung bei der Beschreibung von thermodynamischen Prozessen für Technologen [BVH10].

Bewertung: Die Spezifikation der SysML ist allgemeingültig formuliert und dient als Modellierungssprache für komplexe Systeme in unterschiedlichen industriellen Bereichen, wie Automobil, Luft- und Raumfahrt, Telekommunikation, Maschinen- und Anlagenbau etc. Durch diese Vielschichtigkeit begründet, unterstützt die SysML mehrere Modellierungsansätze (strukturiert, objektorientiert). Des Weiteren sieht die Spezifikation in Analogie zur UML keine vollständige Beschreibung der Semantik vor, welche disziplinspezifisch zu definieren ist [VHBKF11], [WRK10]. Ebenso muss die Anwendbarkeit der Modellierungssprache je nach Anwendungsgebiet getrennt bewertet werden, weil gemäß Spezifikation keine Methodik zur Anwendung der Sprache definiert wird.

3.3.2 Modellierung reaktiver Systeme

Systeme lassen sich in drei Klassen einteilen: transformationell, interaktiv und reaktiv [Sch05]. Transformationelle Systeme überführen Eingabe in Ausgaben, wobei der Benutzer solcher Systeme keinen weiteren Einfluss auf die Verarbeitungskette nehmen kann. Das Verhalten von interaktiven Systemen wird hochgradig durch die Intention des Benutzers beeinflusst. Reaktive Systeme zeichnen sich durch ihre hohe Interaktion und Abhängigkeit von der Umwelt aus, die durch physikalische Größen (bspw. Temperatur, Feuchtigkeit, Vibrationen etc.) und Gesetze auf das System einwirken.

3.3.2.1 Modellbildung für formale Verifikation

In dem Artikel von Kock et al. [KKBB08] wird ein MIL/SIL und PIL Test vorgestellt. Der besondere Fokus liegt dabei auf der formalen Verifikation mit Hilfe eines Model Checkers. Das zu verifizierende System wird als Zustandsautomat definiert, der auf bspw. Erreichbarkeitsanalysen hin untersucht werden kann. Die automatischen Analysen werden auf Codeebene (Division durch Null etc.) durchgeführt.

Bewertung: Die formale Systemspezifikation erfordert eine hohe Modellqualität und bietet somit nur eine unzureichende Unterstützung für die grobgranulare Spezifikation in frühen Phasen. Eine formale und textbasierte Spezifikation für Model Checker erfordert ein tiefgreifendes Modellierungsverständnis und kann nicht als interdisziplinäres Kommunikationsmodell eingesetzt werden. Die Integration von Fehlern ist

grundsätzlich möglich, wird jedoch nicht näher erläutert. Verschiedene Sichten auf ein System können durch vernetzte Zustandsautomaten realisiert werden, wird jedoch in dem vorgeschlagenen Ansatz nicht erläutert.

In [HV05] wird ein Framework vorgestellt, das die Modellierung und Verifikation (mit Hilfe eines Model-Checkers) von NCES (Net Condition/Event Systems) basierten Modellen ermöglicht. Es wird die Modellierung von verteilten Systemen mit Hilfe von graphischen, ausführbaren Modellen, die modular kombinierbar sind, vorgestellt. Zur Beschreibung des Verhaltens wird der NCES als modulare Erweiterung des SNS (Signal/Net System) Modellierungsformalismus verwendet, mit dem Module über Signalflüsse miteinander verbunden werden können. Die zu verifizierenden Ausdrücke müssen dabei bspw. in temporaler Logik formuliert werden. Die visuelle Spezifikationssprache SNS wird in [VB08] beschrieben.

Bewertung: Für die formale Verifikation werden präzise Modellinformationen benötigt. Zur Spezifikation dient der textuelle SNS Modellierungsformalismus. Die fehlende graphische Repräsentation erfüllt nicht die Anforderungen der Entwicklungsingenieure. Der Ansatz beinhaltet kein Sichtenkonzept und besitzt keinen Mechanismus zur Integration von Fehlermodellen.

In [EM12] wird ein modellbasierter Entwicklungsansatz für das Design komplexer (verteilter) Automatisierungssysteme vorgestellt. Zur Definition domänenspezifischer Modelle für 3+1 Sichten werden die drei Industriestandards CAEX (Computer Aided Engineering eXchange), PLCopen und MathML zur mathematischen Beschreibung von Constraints verwendet. Durch den Einsatz eines Model-Checkers kann die Korrektheit der Spezifikation nachgewiesen werden. Mit Hilfe der Modelltransformation können anschließend Designfehler identifiziert werden.

Bewertung: In dem Beitrag liegt der Fokus auf der Definition der domänenspezifischen Sprache zur Konstruktion eines domänenübergreifenden Modells. Kompositionsregeln werden in MathML formuliert. Fehlermodelle sind nicht Bestandteil des übergreifenden Modells.

3.3.2.2 Modellierung mit Modelica

Modelica ist ein von der Modelica Association bereit gestelltes Werkzeug und Modellierungssprache zur objektorientierten, gleichungsbasierten Beschreibung komplexer Systeme [Mod10]. Zur Steigerung der Modellierungseffektivität gibt es eine Vielzahl von Modellbibliotheken für u.a. analoge, digitale, mechanische etc. Komponenten. In Kombination mit Simulationsumgebungen wie bspw. Dymola können Modelica Modelle als Simulation zur Ausführung gebracht werden [Liu14]. Es findet nicht zuletzt wegen des hohen Detaillierungsgrades der Modelle vorwiegend in der Automobilindustrie Anwendung. In vereinzelten Industriezweigen des Maschinen- und Anlagenbaus, die hohe Sicherheitsanforderungen erfüllen müssen, wie bspw. Kraftwerksanlagen kommt Modelica zum Einsatz.

Johnson stellt einen formalen Ansatz zur Kopplung von SysML und Modelica Modellen zur Modellierung kontinuierlichen Systemverhaltens vor [JJPB07]. Durch eine bidirektionale Sprachabbildung können Systemdynamikmodelle durch Differen-

tialgleichungen in SysML beschrieben werden. Die beiden Modelle weisen in ihren strukturellen Eigenschaften hohe Ähnlichkeiten auf, so dass eine Zuordnung der Modelldeklaration direkt umsetzbar ist. Die Modellierungsentität für Strukturelemente in Modelica bilden Klassen, die die Objektstruktur und das Verhalten beschreiben und dadurch auf SysML Blöcke abgebildet werden. Diese Blöcke kapseln die internen Eigenschaften, deren Verhalten und Interaktionsmechanismen zu anderen Blöcken. Das gleichungsbasierte Verhalten wird demnach zur Beschreibung der Beziehung verwendet. In SysML wird dies in Form von Constraints (mathematische Ausdrücke) modelliert, indem das Modell um das Parameterdiagramm erweitert wird. Die darüber modellierten Constraints werden zum Austausch von Energie und Signalen über Konnektoren miteinander verknüpft.

Bewertung: Aufgrund der hohen konzeptionellen Modellierungsähnlichkeit zwischen Modelica und SysML werden die wesentlichen Anforderungen wie dem interdisziplinären Charakter, der graphischen Beschreibungssprache sowie einer identischen Modellierungsgranularität Rechnung getragen. Das Parameterdiagramm wird für die Integration der Differentialgleichungen zur Verhaltensbeschreibung verwendet. Eine explizite Berücksichtigung von Fehlermodellen bzw. Fehlverhalten ist nicht Bestandteil des Ansatzes.

Frey und Liu präsentieren einen Modellierungs- und Simulationsansatz mit Modelica zur Analyse des Zeitverhaltens vernetzter Automatisierungs- und Regelungssysteme [FL09]. Es wird eine Bibliothek aufgebaut, die der Struktur realer automatisierungstechnischer Komponenten entspricht (starke Kopplung). Dabei erfolgt eine Einteilung in drei Domänen: Kommunikationsnetzwerk, Informationsverarbeitung (Eingebettete Steuerung) und der steuernde Prozess. Der Informationsaustausch zwischen den Domänen wird durch drei Typen modelliert (netzwerkbezogene Datenpakete, Nutzdatenaustausch, physikalische Werte). Durch geeignete Abstraktionsmechanismen wird ein effizientes Simulationsverhalten ermöglicht. Die Validierung wurde im offenen und geschlossenen Kreis anhand eines Automatisierungssystems im Labor durchgeführt. Mit Hilfe der Bibliotheksorganisation entsteht eine aufwandsarme Möglichkeit der Modellbildung. In [LFF12] wird der Ansatz auf CPS erweitert.

Bewertung: Durch die Aufteilung in drei Domänen wird ein Sichtenkonzept unterstützt. Die Simulation durch bspw. Dymola ist echtzeitfähig ausführbar, wenn ein hinreichend hoher Abstraktionsgrad der Modellierungsgranularität aller Komponenten sichergestellt ist. Die Integration von Fehlermodellen ist nicht berücksichtigt oder angedacht.

3.3.2.3 Modellierung mit Matlab/Simulink

Matlab ist ein Softwaresystem zur Lösung und Visualisierung mathematischer Probleme und findet verstärkt Anwendung in den Bereichen Regelungstechnik, Bildverarbeitung, Modellbildung und Verifikation. Simulink ist eine Erweiterung für Matlab zur blockbasierten Modellierng technischer bzw. physikalischer Systeme [ABRW09].

In dem Artikel von Gühmann [GÖ2] wird eine Modellbildung für einen automatisierten Testprozess für HiL Simulationen beschrieben und am Beispiel einer Getriebesteuerung für Stufenautomaten vorgestellt. Von besonderer Bedeutung sind

echtzeitfähige Simulationen für das gesamte Modell, welches neben den direkt notwendigen Informationen für die Funktionsüberprüfung auch ein Fahrermodell sowie unterschiedliche Streckenprofile und Umweltbedingungen vereinen muss. Das Simulink Gesamtmodell unterstützt die Modellbildung auf System- und Subsystemebene (Einteilung in Baugruppen) und bietet die Möglichkeit, mathematisch beschriebene Fehler unterschiedlicher Arten (u.a. Kommunikationsfehler, Spannungsfehler, mechanische Fehler) bei den automatisierten Tests zu erzwingen.

Bewertung: Das beschriebene Vorgehen wird für die Automobilindustrie vorgestellt. Die graphische Beschreibungsmöglichkeit durch Simulink erfüllt die Akzeptanzkriterien der Steuerungsprogrammierer, unterstützt jedoch nicht den interdisziplinären Charakter. Die vorgeschlagene Modellierungsgranularität unterstützt die Modellbildung von der Komponenten- bis zur Systemebene. Fehler können explizit im Modell erzwungen werden, jedoch wird kein Vorschlag präsentiert, in welcher Form Fehlermodelle explizit in das Gesamtmodell integriert werden.

Vulinovic und Schlingloff [VS05] präsentieren einen modellbasierten Fehlerinjektionsansatz bei der funktionalen Entwicklung automobiler Steuerungssoftware in Matlab/Simulink. Der Ansatz berücksichtigt dabei mögliche Fehlerszenarien auf vier verschiedenen Abstraktionsebenen: Funktion, Block, Quelltext und Gatter. Auf Funktionsebene können Komponenten- und Kommunikationsfehler simuliert werden. In der Blockebene wird zwischen sog. Mutanten und Saboteuren unterschieden. Mutanten verfälschen die beabsichtigte Funktionalität (Permutation, Zufällige Auswahl) und Saboteure stören die Kommunikation zwischen Blöcken (Trennen von Verbindungen, Signalverstärkung, Rauschen). Auf Codeebene stehen die Möglichkeiten einzelne Bits in Maschineninstruktionen zu kippen. Durch Modifikationen in den Speicherzellen und der HDL Beschreibung können Fehler auf Gatterebene injiziert werden.

Bewertung: Der Ansatz von Vulinovic und Schlingloff zielt auf die gezielte Manipulation von Funktionsverhalten gemäß vorgegebener Wahrscheinlichkeitsverteilungen ab. Zur Modellbildung wird die Matlab/Simulink Werkzeugkette verwendet. Es werden keinerlei Modellierungsvorgaben für die Fehlerberücksichtigung angegeben. Des Weiteren ist das Verfahren nicht für Automatisierungssysteme und einer Sichtenunterstützung ausgelegt.

3.3.3 Domänenspezifische Systemmodellierung

Die Auslegung von Führungsfunktionen für Fahrzeugantriebe wird zur Konsistenzsicherung von Systemmodell und Risikoanalyse miteinander gekoppelt [Rin02]. Für die qualitative und quantitative Risikoanalyse werden System-FMEA und System-FTA eingesetzt. Aus dem Systemmodell wird die Systemstruktur der Risikoanalyse abgeleitet. Die Verhaltensbeschreibung erfolgt mit Hilfe von UML Aktivitäts- und Zustandsdiagrammen. Das vorgeschlagene Verfahren zur Integration der Risikoanalyse dient als Basis für die sicherheitsbezogene Systemoptimierung im Fehlerfall.

Bewertung: Der Ansatz adressiert die Kopplung von Systemmodell und Risikoanalyse für fahrzeugrelevante Sicherheitsaspekte. Dadurch erfolgt ein starker Bezug zu automobilbezogenen Anforderungen. Die Methode ist inhärent nicht für die Betrachtung von technischem System, technischer Prozess und AT-System ausgelegt.

3.3.3.1 Domänenspezifische Modellierung mit SysML

Die Beschreibung eines industriellen Automatisierungssystems mit 3+1 Sichten wird von Thramboulidis [Thr10] erläutert. Die Unterteilung des Systems in Sichten folgt dabei der üblichen mechatronischen Aufteilung in die jeweiligen Disziplinen Mechanik, Elektronik und Software. Alle Sichten werden in eine mechatronische Sicht in SysML integriert, welche eine Systemrepräsentation auf hoher Ebene darstellt. Die Komponenten, aus denen sich das System zusammensetzt, werden entweder explizit an eine Disziplin gebunden oder repräsentieren eine mechatronische Komponente. Um die Interaktion zwischen Komponenten auszudrücken, werden zu den disziplinspezifischen Ports sog. mechatronische Ports eingeführt. Diese Sichtendarstellung ermöglicht Experten die relevanten Aspekte des Systems differenzieren zu können. Eine weitere Verwendung der 3+1 SysML view model für das Upgrade von Legacy Systemen wird in [STF11] präsentiert.

Bewertung: Der ganzheitliche Ansatz kombiniert die Modelle der unterschiedlichen an der Entwicklung beteiligten Disziplinen und präsentiert eine übergreifende mechatronische Sicht auf IEC 61499 Systeme. Die Modellierungsgranularität des Systems wird dabei nicht im Detail adressiert. Es geht primär um die Sichten für das MIM (Model Integrated Mechatronics) Paradigma. Eine Integration von Fehlermodellen bzw. Fehlverhalten ist nicht vorgesehen.

Bassi et al. [BSBF10] setzen SysML zur abstrakten Beschreibung komplexer Produktionssysteme ein. Um das während des Entwicklungsprozesses solcher Systeme generierte hohe Datenaufkommen bewältigen zu können, wird ein hierarchisches Modell mit drei zueinander in Verbindung stehenden Abstraktionsebenen eingeführt. Auf der Detailebene wird das Verhalten jedes Subsystems präzise modelliert. In der darüber liegenden globalen Ebene werden die Subsysteme zu einem Gesamtverhalten über Schnittstellen miteinander gekoppelt. Auf oberster Ebene werden die Anforderungen an das System sowie deren funktionale Aspekte beschrieben. Die Spezifikation auf der obersten Ebene erfolgt in SysML mit dem Anforderungs-, Blockdefinitions- und internen Blockdiagramm. Die darüber definierten Module werden auf die Software und die Physik abgebildet. Der Prozess der iterativen Verfeinerung, Komposition und Validierung wird anhand eines Aktivitätsdiagramms in dem Artikel beschrieben. An dem Beispiel einer Verpackungsmaschine werden der Designprozess und die Validierungsregeln näher dargestellt.

Bewertung: Die präsentierte Modellierungsmethodik beschreibt das Vorgehen bei der high-level Systembeschreibung in den Aspekten der Anforderungen, Funktionalität und Systemstruktur in SysML. Dabei werden keine expliziten Vorgaben zur Modellierungsgranularität eingeführt. Eine Integration von Fehlermodellen wird nicht berücksichtigt.

In *Domänenspezifische Modellierung für automatisierungstechnische Anlagen mit Hilfe der SysML* beschreiben die Autoren eine entwicklungsphasen- und disziplinübergreifende Unterstützung für Anlagenentwickler [SW09]. Der Ansatz stützt sich auf die Verwendung des drei Sichtenkonzepts nach Lauber und Göhner [LG99]. Diese Sichten umfassen das technische System, den technischen Prozess und das Automatisierungs-

system. Das technische System, d.h. die Verschaltung von mechanischen Komponenten, werden mit dem internen Blockdiagramm (IBD) der SysML modelliert. Der technische Prozess beschreibt die zeitliche Veränderung des zu bearbeitenden Guts. Mit Hilfe des SysML Aktivitätsdiagramms stellen die Autoren eine Beschreibungssprache vor, die sich in ihrem Abstraktionsniveau besonders für die Beschreibung eines Prozessablaufs (Rezept) durch Technologen (Prozessingenieur) eignet. Das konkrete Verhalten der Komponenten des technischen Systems wird mit Hilfe des Zustandsdiagramms beschrieben. Dabei werden physikalische Zustände (bspw. Winkelstellungen einer Weiche) als Modellzustände dargestellt. Das Automatisierungssystem, bestehend aus Steuerung, Netzwerktechnik und Aktoren, wird ebenfalls mit Hilfe des internen Blockdiagramms beschrieben.

Bewertung: Der präsentierte Ansatz erfüllt die zentralen Aspekte der graphischen Modellierung und unterstützt im interdisziplinären Umfeld. Die integrale Berücksichtigung der Sichten ermöglicht eine zielgerichtete Adressierung der wesentlichen Aspekte eines Automatisierungssystems. Der Ansatz bietet keine Berücksichtigung von Fehlern bzw. Fehlverhalten.

3.3.3.2 Domänenspezifische Modellierung mit CAD Systemen

Zäh et al. schlagen eine Erweiterung des CAD-Modells zur Simulation des Materialflusses einer Produktionsanlage vor [ZSL08]. Ein maßgebliches Kriterium für die Aussagekraft von Simulationen stellt die Berücksichtigung des Materialflusses dar. Das reduzierte CAD Datenmodell wird um ein Physik- und Kinematikmodell erweitert. Die Erstellung des gesamten Simulationsmodells wird sequentiell durch Anreicherung des Geometriemodells um Physik- und Kinematikmodell verfeinert. Das unveränderte Geometriemodell wird durch Tesselierung in ein polygonales Oberflächenmodell überführt, das durch Vereinfachungen für die Simulation verwendbar wird. Dieses wird anschließend um physikalische Eigenschaften wie Reibungseffekte angereichert und in einem kinematischen Modell zusammengefügt. Als Simulationsumgebung wird ein eigens entwickeltes System eingesetzt.

Bewertung: Der Ansatz beschreibt ein Vorgehen zur Anreicherung vorhandener CAD Modelle um den Materialfluss, für aussagekräftige Simulationen von Produktionsanlagen. Dabei werden im speziellen physikalische und kinematische Eigenschaften hinzugefügt. Eine explizite Betrachtung von Fehlern ist nicht Bestandteil des präsentierten Ansatzes.

3.3.4 Gesamtbewertung Systemmodellierung

Die in diesem Kapitel betrachteten Ansätze zur Systemmodellierung sind in der Tabelle 3.2 zusammengefasst dargestellt und gemäß den eingeführten Kriterien bewertet. Die tabellarische Übersicht stellt eine kompakte Bewertung der prosaischen Bewertung aller im vorangegangenen Kapitel analysierten Ansätze dar. Dabei wird deutlich, dass bislang kein Ansatz für die Modellierung von technischen Systemen im Maschinen- und Anlagenbau existiert, der Fehlermodelle explizit im Systemmodell integriert. Die einzigen Ansätze, die eine teilweise Integration vorsehen, werden im Automotivebereich angewandt.

Tab. 3.2: Bewertungsübersicht der Systemmodellierung gemäß den Kriterien aus Kapitel 3.1.1

Verfahren / Quelle	Unterstützung von Sichten	Modellierungsgranularität	Graphische Beschreibungssprache	Integration von Fehlern	Maschinen- und Anlagenbau	IEC 61131-3
Matlab / [Gö2]	○	●	●	◑	○	○
Formal / [KKBB08]	◑	●	○	◑	○	○
Modelia / [JJPB07]	◑	●	●	○	◑	○
Matlab / [VS05]	○	◑	●	◑	○	○
SysML / [Thr10]	●	◑	●	○	●	○
SysML / [BSBF10]	○	◑	●	○	●	◑
Modelica / [FL09]	●	◑	●	○	●	◑
Formal / [HV05]	◑	●	○	○	●	○
Formal / [EM12]	◑	◑	◑	○	●	●
CAD / [ZSL08]	○	●	◑	○	●	○
SysML / [SW09]	●	●	●	○	●	●
DS[1] / [Rin02]	◑	●	●	●	○	○
Legende: ●: Vollständig erfüllt ◑: Teilweise erfüllt / nicht feststellbar ○: Nicht erfüllt						

[1] Domänenspezifisch

3.4 Testverfahren für Automatisierungssoftware

In diesem Kapitel werden die Grundlagen zu etablierten Testverfahren präsentiert und die darauf aufbauenden Ansätze gemäß den in Kapitel 3.1.2 eingeführten Kriterien bewertet.

3.4.1 Funktionale Testverfahren

Funktionale Testverfahren werden gängigerweise auch als Black Box Testverfahren bezeichnet, da bei der Testdurchführung das zu testende System (Testobjekt) an seinen Schnittstellen zur Umgebung, d.h. den Eingangs- und Ausgangsgrößen betrachtet wird. Dabei wird das Testobjekt mit den ausgewählten Eingangsdaten (Testdaten) stimuliert und das Verhalten durch einen Vergleich der erwarteten mit den tatsächlichen Ausgangsdaten verifiziert. Erfolgt dabei eine Abweichung zur Spezifikation, führt dies zum Fehlschlagen des Tests. Die Auswahl der Testdaten bei funktionalen Testverfahren erfolgt aus dem spezifizierten Verhalten und lässt dabei die interne Struktur der Realisierung außer Acht. Dies kann folglich dazu führen, dass beim Testfalldesign die Auswahl der Testdaten nicht zielgerichtet gewählt werden und Fehler in der Umsetzung nicht aufgedeckt werden können. Typische Fehler, die somit quasi nur zufällig aufgedeckt werden können, sind Datentypkonvertierungen auf einen engeren Wertebereich, was die Gefahr eines Überlaufs nach sich zieht.

3.4.1.1 Grundlegende Verfahren

Zur Reduktion der Komplexität werden Verfahren verwendet, die eine Partitionierung der Testdaten vornehmen und dadurch die Anzahl der Testläufe reduzieren. Mit Hilfe der Äquivalenzklassenbildung [DRP99] lassen sich Eingangsdaten in funktional ähnliche Klassen einteilen, so dass anschließend jeweils ein repräsentativer Kandidat jeder Klasse zum Test ausgewählt wird, und das Testergebnis stellvertretend für alle weiteren Kandidaten der jeweiligen Klasse gilt. Der Bestimmungsprozess der Äquivalenzklassen stellt dabei einen entscheidenden Faktor in der Aussagekraft der anschließenden Testergebnisse dar. Dieser Prozess ist hochgradig manuell und hängt stark von der Systemkenntnis und der Erfahrung des Testdesigners ab. Zur Unterstützung der Äquivalenzklassenbildung kann eine Ursache Wirkungsanalyse [DRP99] herangezogen werden. Dabei werden die möglichen Eingangsparametervariationen mit den erwarteten Systemverhalten gegenübergestellt und besonders kritische Variationen als relevant markiert. So stellt bei der Berechnungsvorschrift zur Lösung quadratischer Gleichungen $x_{1/2} = \frac{-b \pm \sqrt{b^2-4ac}}{2a}$ der Fall einer Division durch 0 ($a_1 = \{0\}$) eine zu testende Wirkung dar. Eine Erweiterung der Äquivalenzklassenbildung ist der Test auf Basis einer Grenzwertanalyse [DRP99]. Dabei werden die Werte an den Grenzen als kritische Eingangsparameter betrachtet, da je nach Anwendungsfall diese Werte signifikante Verhaltensextrema provozieren können. Bei der reinen datentypbezogenen Betrachtung, bspw. vorzeichenbehaftete 32-Bit Ganzzahl, wären dies die minimal (-2^{31}) und maximal ($2^{31} - 1$) darstellbaren Zahlen.

Ein weiteres Partitionierungsverfahren stellt die von Dailmer-Benz entwickelte Klassifikationsbaummethode [GG06] als eine Erweiterung der Category-Partition Method

[OB88] dar, die verstärkt zur Testfall- und Testdatenermittlung im Automobilbereich eingesetzt wird. Dabei wird die funktionale Spezifikation eines zu testenden Systems analysiert und alle testrelevanten Aspekte ermittelt. Durch eine geeignete Kombination der Testwerte werden Testfälle erstellt. Abbildung 3.7 zeigt einen exemplarischen Klassifikationsbaum für ein Antibremsblockiersystem (ABS), das im Falle einer Vollbremsung den Bremsschlupf im optimalen Bereich halten soll. In dem Beispiel sind die zwei Klassifikationen Geschwindigkeit und Fahrbahn als zentrale Parameter für die Testdurchführung relevant. Die darunter aufgefächerten Werte (Klassen) repräsentieren die als testrelevant (typische, praxisrelevante Szenarien) erachteten Parameter. Durch die Kombination aller von der Klassifikation verschiedenen Klassen, ergeben sich die Testfälle. Das ABS System muss demnach im Testfall 1 bei Tempo 30 km/h auf einer eisglatten Fahrbahn und im Testfall 3 bei Trockenheit aus dem Tempo 130 km/h getestet werden. Die Klassifikationsbaummethode wurde für eingebettete Systeme derart erweitert, dass die Testfälle als einzelne Testsequenzen interpretiert werden können und so auch zeitliche Verläufe abbildbar werden [Bro02]. In Analogie zur Äquivalenzklassenbildung ist auch bei dieser Methode die Kreativität des Testers der entscheidende Faktor zur Ermittlung der Aussagekraft der Testfälle und -daten. Bei besonders komplexen Systemen kann eine quantitative Zunahme der Klassifikationen schnell zu einem unübersichtlichen und somit schwer handhabbaren Baum führen.

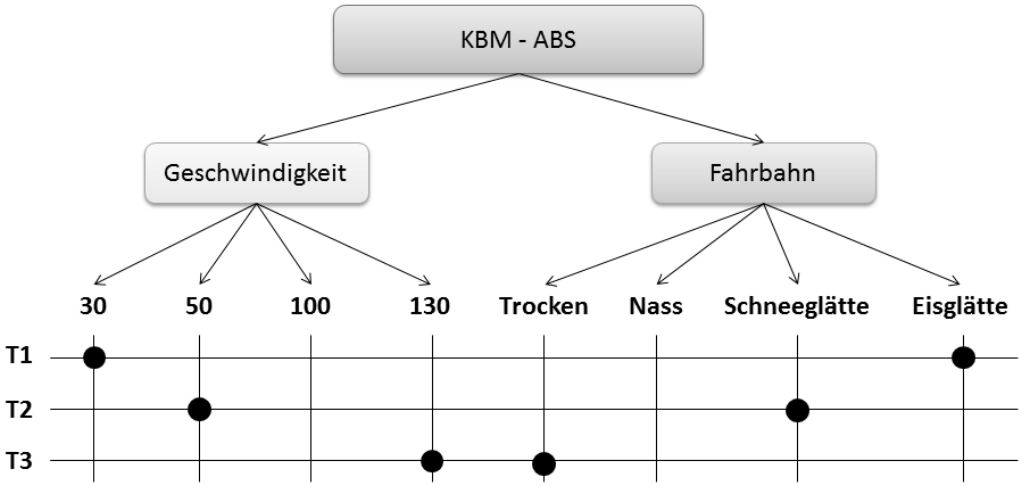


Abb. 3.7: Drei exemplarische Testfälle für ein ABS System zur Erläuterung der Klassifikationsbaummethode

Time Partition Testing ist eine Methode für den Test kontinuierlichen Verhaltens eingebetteter Systeme [Leh04]. Es umfasst die Testmodellierung, Testdurchführung, Testauswertung und Testdokumentation. Die Testmodellierung erfolgt in einer graphischen formalen Automatennotation. Die Modellierung der Testfälle erfolgt dabei als phasenbasierter Ablauf. Die TPT Testmodellierung vereint die Testfälle mit den Testdaten. Die daraus entstehenden Testfälle unterscheiden sich bei der Variantenbildung von Zuständen, Transitionen oder Pfaden im Detail.

3.4.1.2 Ansätze funktionaler Testverfahren

Hametner et al. [HKVH11] präsentieren einen Ansatz zur automatischen Testfallgenerierung für industrielle Automatisierungssysteme. Die Testfälle zur Verifikation des Verhaltens werden aus einer Spezifikation heraus generiert. Die Verhaltensspezifikation wird dabei in einem Anlagenverhaltensmodell und einem Steuerungsverhaltensmodell unterteilt. Ersteres umfasst alle relevanten Aspekte des physikalischen und mechanischen Verhaltens der Systembestandteile inklusive aller zeitlichen Aspekte. Das Steuerungsverhaltensmodell beschreibt das logische Verhalten aller Komponenten, d.h. den zur Funktionserfüllung der Anlage notwendigen Einzelaktionen. Zur Beschreibung der Verhaltensmodelle dienen UML Zustandsdiagramme, aus denen in Form von Modelltransformationen Testfälle als FB durch den Einsatz der Transformationssprache Xtend generiert werden. Der Ansatz wird am Beispiel einer Sortieranlage demonstriert.

Bewertung: Der allgemeine Ansatz eignet sich sowohl für IEC 61499 als auch IEC 61131 basierte industrielle Applikationen. Der automatisierte Ansatz ermöglicht eine zielgerichtete, aufwandsarme Ableitung von Testdatensätzen, die jedoch aus einer Spezifikation und ohne weitere Restriktion keine Aussagekraft über die Güte der erzeugten Testfälle mit deren verbundenen Testdaten zulässt. Der Ansatz berücksichtigt keine fehlerbezogenen Testdaten.

Krause et al. beschreiben ein Verfahren zur automatischen Generierung von Testfällen aus einer Spezifikation für eingebettete Systeme [KHD08]. Die Spezifikation beschreibt das Ein-/Ausgangsverhalten eines modellierten Systems mit Hilfe von UML Zustandsdiagrammen. Die Generierung der Testfälle erfolgt dabei in zwei Schritten. Das UML Zustandsdiagramm (Zustand, Transition etc.) wird auf ein ESTPN (Extended Safe Place/Transition Net) abgebildet. Diese Systemspezifikation dient als Basis zur Testfallgenerierung, indem Methoden der fundierten Petri-Netz Theorie angewandt werden. Das daraus entstandene ESTPN wird anschließend aufgefaltet und jede Alternative des vollständigen Präfix generiert. Eine optimierte algorithmische Vorschrift zur Berechnung von McMillan's *unfolding algorithm* wird in [ERV02] präsentiert. Jede Alternative repräsentiert einen Testfall. Die Betrachtung der ereignisbasierten Kommunikation zwischen mehreren Zustandsmaschinen zur Testfallgenerierung für die Spezifikation verteilter Systeme ist angedacht.

Bewertung: Der automatische Ansatz erzeugt Testfälle aus einer Verhaltensspezifikation. Dabei erfolgt weder eine Reduktion/Selektion der Testdaten, noch besteht ein konkreter Bezug zu fehlerbezogenen Kriterien.

In dem Artikel *Deriving Input Partitions from UML Models for Automatic Test Generation* beschreiben die Autoren einen Ansatz zur Eingabedatenpartitionierung auf Basis von UML Modellen [WS08]. Unter Verwendung der UML Klassendiagramme, UML Zustandsdiagramme und der Object Constraint Language (OCL) wird der anschließende Testcode automatisch generiert. Durch eine Auswertung der OCL Ausdrücke und deren Zusammenhänge wird eine Grenzwertanalyse der Eingabedaten durchgeführt und dadurch die testrelevanten Werte ermittelt. Eine generische Sortiermaschine dient als Demonstrationsbeispiel des Ansatzes.

Bewertung: Durch die automatische Grenzwertanalyse erfolgt eine Reduktion auf

anwendungsfallspezifische, aussagekräftige Testdaten. Der Ansatz verwendet eine UML Spezifikation und bezieht sich nicht auf fehlerrelevante Kriterien. Die Testdatengenerierung ist auf ereignisbasierte Systeme ausgelegt und nicht für IEC 61131-3 basierte Systeme.

Schwarzl und Peischl schlagen einen Ansatz zur Testfallgenerierung vor, der auf symbolischen Transitionssystemen basiert [SP10]. Testfälle werden aus Verhaltensspezifikation von UML Zustandsdiagrammen, basierend auf deren Kommunikationsstruktur abgeleitet. Die ausgehenden Transitionen der Zustandsautomaten werden zufällig durchlaufen, wodurch Pfade systematisch aufgedeckt werden. Die dazu notwendigen Modelltransformationen (bspw. Model Flattening) werden im Detail erläutert. Anhand einer Blinkeranwendung im automobilen Umfeld wird der Ansatz veranschaulicht.

Bewertung: Der Ansatz zeigt ein weiteres Verfahren der Testfallgenerierung aus UML Zustandsdiagrammen. Das systematische Verfahren der Generierung besitzt keine Differenzierungsmöglichkeit, um aussagekräftige (Zielaspekt) Daten für Testfälle zu generieren. Zur Ableitung wird eine zu erstellende Spezifikation benötigt, die keine Fehlerzweige berücksichtigt.

In dem Beitrag *Automatische Testfallgenerierung mittels Model-Checking für Steuerungsprogramme* wird ein Ansatz zur Testfallgenerierung auf Basis formaler Modelle präsentiert [KWV10]. Die im Maschinenbau übliche Beschreibungsform des Weg-Zeit-Diagramms wird zusammen mit einer I/O Liste und dem IEC 61131-3 Steuerungsprogramm in ein Umgebungsmodell und ein Modell der Steuerungssoftware transformiert. Die Modelle sind zeitbehaftete Automaten für den Model-Checker UPPAAL. Mit Hilfe des Tools UPPAAL Cover werden basierend auf angegebenen Testparametern (bspw. Abdeckungsgrad) die Automatennetzwerke automatisch analysiert und Testsequenzen generiert, die anschließend mit dem Werkzeug UPPAAL-Tron zur Ausführung gebracht werden können. Der Ansatz wird an einem Model einer Laboranlage demonstriert.

Bewertung: Der vorgeschlagene Testgenerierungsansatz erzeugt gemäß den transformierten Automaten und den Abdeckungskriterien eine Vielzahl an Testfällen, so dass kein Zusatzkriterium zur Einschränkung auf aussagekräftige Testdaten berücksichtigt wird. Die Generierung erfolgt aus dem Umgebungs- und Steuerungsprogrammmodell. Eine direkte Berücksichtigung von Fehlern ist nicht vorgesehen.

Ein weiterer modellbasierter Ansatz zur Testfallgenerierung aus UML Zustandsdiagrammen wird in [CTF01] vorgeschlagen. Zur Generierung der Testfälle wird die UML Spezifikation mit einer statistischen, funktionalen Testmethode ausgewertet. Dieses Verfahren generiert zufällig Eingabedaten, die gemäß der zugewiesenen Wahrscheinlichkeitsverteilung angenommen werden können. Der Wertebereich und die damit einhergehende Wahrscheinlichkeitsverteilung muss für die Eingabedaten empirisch ermittelt werden. Die generierten Testfälle müssen einem angegebenen Abdeckungskriterium (Transitionsabdeckung) der Zustandsdiagramme genügen. Der Ansatz wird am Beispiel eines Avioniksystems demonstriert.

Bewertung: Das präsentierte Testverfahren verwendet UML Zustandsdiagramme als Spezifikation und leitet Testeingabedaten auf Basis einer Wahrscheinlichkeitsverteilung ab, die vorab empirisch ermittelt werden muss. Dies stellt einen weiteren

Aufwandsfaktor dar. Außerdem ist der Ansatz nicht für IEC 61131-3 basierte System vorgesehen und unterstützt keine Fehleraspekte.

Im Forschungsprojekt ZuMaTra wurde ein Verfahren zum automatisierten Test von Fehlerbehandlungsroutinen erarbeitet [ZuM15]. Für die Generierung der Testfälle dienen durch Fehleroperatoren angereicherte Weg-Zeit-Diagramme, die das Gutverhalten der Maschine beschreiben. Die Fehleroperatoren beschreiben eine Vielzahl theoretisch möglicher Abweichungen. Es fehlt jedoch ein diskriminierendes Kriterium zur Selektion der Fehlercharakteristik. Denn häufig ist die Bedeutung und praktische Relevanz eines Fehleroperators (bspw. Nicht Einhaltung eines Intervalls) nicht bekannt. Das Verhalten der Maschine im Fehlerfall muss bekannt und spezifiziert sein.
Bewertung: Das erarbeitete Verfahren stützt sich auf eine Spezifikation der Gutfunktionalität. Ein wesentlicher Anspruch ist, nicht gegen eine Spezifikation zu prüfen, sondern das tatsächliche Verhalten zu überprüfen. Zyklische Steuerungsprogramme werden unterstützt, da es sich um ein Verfahren für IEC 61131-3 basierte Applikationen handelt. Durch das fehlende Testkriterium werden einerseits eine Vielzahl theoretischer Tests generiert, deren Aussagekraft für die praktische Relevanz nicht eingeordnet werden kann.

Ein Ansatz zur Testfallgenerierung aus vollständigen Systemanforderungsmodellen wird in [Kel09] eingeführt. Anforderungen müssen dabei mit Hilfe der formalen Notation SpecTRM-RL, einer Anforderungsspezifikationssprache entwickelt am MIT, spezifiziert werden. Als Grundlage für die Ableitung der Testfälle werden sog. Szenarien identifiziert, die jeweils eine Menge an Bedingungen umfasst, die denselben Systemzustand zur Folge haben. Eingabedaten werden anschließend generiert, so dass diese die jeweiligen Szenarien erfüllen. In dem Beitrag werden Algorithmen zur Berechnung der Tests auf deren Anwendbarkeit miteinander verglichen. Drei der vier ausgewählten Algorithmen zeigen ein lineares Wachstum mit der Systemkomplexität, d.h. die Anzahl der generierten Testfälle wächst in etwa linear mit der Systemgröße an.
Bewertung: Das vorgestellte Verfahren ermöglicht die Generierung von Testfällen aus Anforderungsmodellen. Diese müssen jedoch formal spezifiziert werden. Die Testdatenreduktion erfolgt aufgrund von Szenarien, die in den Anforderungen festgehalten wurden. Da im Rahmen der Umsetzung ein Drift gegenüber der Spezifikation erfolgen kann, können die ausgewählten Szenarien nicht mit der Realität übereinstimmen. Der Ansatz ist allgemein gehalten und berücksichtigt Fehler nicht explizit.

3.4.2 Strukturelle Testverfahren

Strukturelle Testverfahren zielen auf die Auswertung der internen Facetten eines Systems ab und werden auch als White Box Testverfahren bezeichnet [DRP99]. Derartige Verfahren können Fehler aufdecken, die über eine rein funktionale Analyse nur schwer identifizierbar sind. Da strukturelle Verfahren die Kenntnis der inneren Zusammenhänge eines Systems analysieren, werden sie zumeist von Entwicklern im Rahmen von Unit Tests eingesetzt, wohingegen funktionale Verfahren besonders im System- und

Abnahmetest ihre Anwendung finden. Eine Kombination von zustandsbasiertem und strukturiertem Testen wird in [MBLDP11] verglichen und bewertet.

3.4.2.1 Grundlegende Verfahren

Zur strukturellen Analyse von Softwarequelltext existiert eine Vielzahl an grundlegenden Verfahren [DRP99], die zu einem Großteil automatisiert über spezielle Testwerkzeuge ausgeführt werden können. Die zentralen Verfahren werden im Folgenden näher erläutert:

- **Anweisungsüberdeckung** C_0[1] Hierbei wird versucht, die Anzahl der ausgeführten (erreichbaren) Anweisungen zu ermitteln. Dadurch kann ein eventuell vorhandener, nicht erreichbarer Code (toter Code) aufgedeckt werden. Bei diesem Analyseverfahren werden die Zweige nicht betrachtet, so dass es zu einer unsystematischen Bewertung des gesamten Quelltextes und u.U. nicht allgemeingültigen Aussagekraft führt. Ein fundamentaler Nachteil dieses Verfahrens ist die fehlende Aussagekraft für Anweisungsüberdeckungen auch auf Objektcodeebene [Mye79]. Es handelt sich dabei um die einfachste Strukturanalysemethode.

- **Zweigüberdeckung** C_1 Die Zweigüberdeckung beinhaltet die Anweisungsüberdeckung und stellt damit ein höheres Analysemaß entsprechend höherer Komplexität dar. Analog zur C_0 Überdeckung werden durch das C_1 Maß nicht erreichbare Zweige aufgedeckt. Wesentliche Nachteile der Zweigüberdeckung sind die Maskierung von Bedingungen, so dass logische Fehler nicht aufgedeckt und dadurch Schleifen unvollständig getestet werden.

- **Pfadüberdeckung** C_2 Die Pfadüberdeckung umfasst alle Pfade eines Quelltextes von einem Ausgangspunkt bis zum Endpunkt, bspw. vom Beginn einer Funktion bis zu deren Ende und zeichnet sich durch eine hohe Fehlererkennungsrate aus. Die Anzahl der Pfade verhält sich exponentiell zur Anzahl der Zweige, weshalb das Verfahren sehr zeitintensiv ist. Erweiterungen des Verfahrens begrenzen zur Komplexitätsreduktion die Anzahl der Schleifendurchläufe [Nta88].

- **Bedingungsüberdeckung** C_3 Zusammengesetzte boolsche Ausdrücke werden bei den vorherigen Verfahren nicht separat ausgewertet. Mit Hilfe der Bedingungsüberdeckung werden alle Subausdrücke unabhängig voneinander analysiert und ausgewertet. Das führt zu ähnlichen Ergebnissen wie die C_1 Abdeckung, die jedoch stärker vom Kontrollfluss abhängig ist. Die Bedingungsüberdeckung stellt den umfangreichsten und aufwändigsten strukturellen Test dar und existiert deshalb auch in drei Ausprägungen zur Skalierung der Durchführungskomplexität.

1 In [DRP99] wird die Anweisungsüberdeckung als C_1 und alle weiteren Überdeckungsmaße entsprechend um eins erhöht bezeichnet. Da es keine feste Definitionen für die Bezeichnung der Abdeckungskriterien gibt, tauchen in der Literatur unterschiedliche Abkürzungen für die jeweiligen Kriterien auf.

- **Datenflussüberdeckung** Bei diesem Verfahren handelt es sich um eine Variation der Pfadüberdeckung. Es wird der Datenfluss in die Testprozedur unter Berücksichtigung der Datenflusscharakteristiken (du-, ur-, dd-Datenflussanomalie) aufgenommen und die entsprechenden Testpfade werden ausgewählt [Bei90]. Die dadurch abgeleiteten Testprozeduren untersuchen Abläufe, die durch Zusammenhänge der Programmvariablen resultieren.

3.4.2.2 Ansätze struktureller Testverfahren

Mit der Verwendung von Constraint Solver Techniken wird in [GBR98] ein Verfahren präsentiert, mit dem Testeingabedaten für eine Untermenge von C Programmen automatisch generiert werden. Ziel ist es, durch die Formulierung von Restriktionen aller Bedingungen zu einer bestimmten Stelle im Quelltext ein Problem zu formulieren, so dass durch den Einsatz eines Constraint Solvers gültige Eingabebelegungen gefunden werden können. Das Verfahren besteht aus insgesamt drei Schritten: der Ausgangsquelltext wird zunächst in die Static Single Assignment (SSA) Form transformiert, eine Darstellung, in der Variablen nur einmal in der gesamten Lebenszeit Zuweisungen erhalten. Anschließend wird eine Menge von Constraints für die zu analysierende Prozedur (Funktion) erzeugt, bevor zuletzt Constraints für die Kontrollflussabhängigkeiten generiert werden. Das Verfahren wird an einem Beispielprogramm mit zahlreichen Kontrollstrukturen (Sequenz, if, if-else, while) evaluiert. Erweiterungen für Zeiger und Gleitkommazahlen sind angedacht.

Bewertung: Der Ansatz ermöglicht die Generierung von Testdaten für die Erreichbarkeit bestimmter Anweisungen im Quelltext. Durch die Formulierung des Kontrollflusses als Constraint Problem werden die Testeingabedaten reduziert. Das Verfahren eignet sich jedoch nicht für zyklische IEC 61131-3 Applikationen, da die Datenabhängigkeiten zwischen den einzelnen Pfaden nicht berücksichtigt werden.

Für Continuous Function Charts (CFC) wird in [AERP04] ein Abdeckungsmaß sowohl für den Daten- als auch den Kontrollfluss definiert, um Integrationstests zu bewerten und komplexe Bedingungen zusammenzuführen, um MC/DC Kriterien anzuwenden. Für den Kontrollfluss werden die maximalen Subgraphen des CFCs identifiziert, die nur aus logischen Operatoren bestehen, woraus die entsprechende komplexe Bedingung errechnet wird. Für den Datenfluss wird der Kontrollflussgraph um def-use Abhängigkeiten erweitert. Der präsentierte Ansatz wird an einem Beispiel mit überwiegend arithmetischen Ausdrücken als auch an einem Beispiel mit logischen Ausdrücken angewendet.

Bewertung: Der Ansatz beschreibt ein Verfahren zur Anwendung eines Abdeckungsmaßes für verschaltete CFC Blöcke. Die Betrachtung der Datenabhängigkeit innerhalb des Kontrollflusses ist ein wesentliches Element für die Testeingabedatengenerierung für zyklischen Softwarebausteine.

Der Beitrag von Majchrzak und Kuchen [MK09] beschreibt eine automatische Testfallgenerierung basierend auf einer Abdeckungsanalyse für Programme, die auf Java ByteCode abgebildet werden. Dieser Zwischencode wird mit Hilfe von Constraint Solving Verfahren symbolisch ausgeführt und die dadurch ausgeführten Pfade

analysiert. Den Nachteil der hohen Testfallgenerierung entgegnen die Autoren damit, dass weitere globale Abdeckungskriterien für den Kontroll- und Datenfluss in Betracht gezogen werden. Der Ansatz wird auf den Bytecode angewendet, da dieser eine höhere Aussagekraft besitzt als der ursprüngliche Ausgangsquelltext vor allen Compileroptimierungsstufen. Für die Lösung der linearen Ungleichungssysteme setzen sie u.a. den Fourier-Motzkin Elimination Algorithmus ein. Die resultierenden Pfade entsprechen individuellen Testfällen. Neben der Betrachtung des Kontrollflusses wird die def-use Abhängigkeit der Daten mit unter Betracht gezogen, d.h. Testfälle können über deren Datenabhängigkeit selektiert bzw. reduziert werden. Das Minimierungsverfahren ist jedoch NP-vollständig[1], so dass die Optimierung über ein Greedy Verfahren umgesetzt wird. Die experimentellen Ergebnisse zeigen, dass die Anzahl der Testfälle mit dem präsentieren Verfahren drastisch reduziert werden können. Teilweise musste die symbolische Ausführung jedoch bereits bei kleinen Programmen (bspw. Quicksort) aus zeitlichen Gründen abgebrochen werden. In künftigen Forschungsarbeiten wollen die Autoren die Elimination bedeutungsloser Testfälle adressieren.

Bewertung: In dem zweistufigen Verfahren werden durch eine dynamische Auswertung der Abdeckungsanalyse potentielle Testfallkandidaten generiert, die durch eine nachgeschaltete Eliminationsmethode derart reduziert werden, dass alle Pfade des Quelltextes mit der minimalen Anzahl an Testfällen abgedeckt werden. Das Verfahren lässt jedoch keinen Rückschluss auf die richtige Wahl und somit die Relevanz der ausgewählten Testeingabedaten zu. Der Ansatz eignet sich für den Java ByteCode und ist nicht auf zyklischen Applikationen anwendbar, da die Datenabhängigkeit lediglich zur Reduktion der Testfälle herangezogen wird.

Jöbstl et al. kombinieren die symbolische Ausführung und SMT Solving (Satisfiability Modulo Theories) zur Generierung von Testfällen [JWAW10]. Der Ansatz stellt einen komplementären Ansatz zu binären Entscheidungsbäumen (BED) dar. Dabei werden sowohl das System als auch die Testziele und Testfälle als symbolische Transitionssysteme nach Rusu et al. [RBJ00] formalisiert. Ein Testfall wird als Transitionssystem ohne interne Transitionen initialisiert. Mit Hilfe des präsentierten adaptierten Algorithmus' zur symbolischen Ausführung werden die Testfälle generiert. Dieser besteht aus sechs Schritten: Vervollständigen des Testziels, Produktberechnung aus Spezifikation und Testziel, deterministischer Schluss des Produkts, Symbolische Ausführung, Testbaumauswahl, Testbaumtransformation. Der in Java umgesetzte Ansatz wurde an dem Session Initiation Protocol (SIP) und einem Conference Protocol evaluiert. Ein Mapping von UML Zustandsdiagrammen auf symbolische Transitionssysteme ist Bestandteil zukünftiger Forschungsaktivitäten.

Bewertung: Der Ansatz beschreibt ein automatisches Verfahren zur Generierung von Testfällen und ergänzt BED basierte Verfahren. Es beruht auf der Spezifikation in symbolischen Transitionssystemen, die in dem Beitrag manuell aus bspw. UML Zustandsdiagrammen überführt werden müssen. Spezielle Aspekte die zur Testgenerierung für IEC 61131-3 Applikationen notwendig sind, werden nicht berücksichtigt.

1 Gemäß dem Millennium Problem P-NP wird vermutet, dass derartige Probleme nicht in polynomialer Zeit lösbar sind.

Ein zentrales Kriterium bei der Durchführung von Tests ist die Erkenntnis, nach wie vielen Tests bzw. Kombinationen von Testdaten ein genügend großes Maß der Testtiefe erreicht ist. Durch den Ansatz eines Datenabdeckungstests soll die kritische Anzahl an Testdaten durch eine 6stufiges Verfahren ermittelt werden [NWM02]. Ziel dieses Ansatzes ist es, eine obere Grenze der Testdatenmenge zu erreichen, so dass alle Fehler des zugrundeliegenden Testmodells aufgedeckt werden können. Dabei ist das Design des Testmodells von entscheidender Bedeutung, die für das spezifische Testobjekt Eingabe- und Ausgabepaare festlegen muss. Dieser Testansatz wird drei Routinen der C++ STL (Standard Template Library) angewendet und zeigt, dass über dieses Verfahren alle integrierten Fehler identifiziert werden konnten und im Vergleich zu Coverage Testing und Random Testing bei geringerer Testdatenmenge und kürzerer Zeit.

Bewertung: Der Ansatz beschreibt ein Verfahren, welches auf C++ Funktionsebene angewendet wird. Das Konzept der Datenabdeckung basiert u.a. auf den folgenden Annahmen über das Testobjekt: deterministisch, keine nicht kontrollierbaren Zufallsvariablen und ein Testorakel kann entwickelt und automatisch ausgeführt werden. Insbesondere das konkrete und gewünschte Verhalten von Maschinen- und Anlagen ist i.d.R. nicht bekannt bzw. schwer formalisierbar, d.h. eine automatische Auswertung ist nicht möglich. Das beschriebene Konzept bietet eine sehr gute Möglichkeit der Testdatenbegrenzung, jedoch müsste für IEC 61131-3 basierte Applikationen auch das grundlegende Verhalten, bspw. in Form einer Simulation, mit betrachtet werden.

3.4.2.3 Symbolic Execution Werkzeuge

Bei der symbolischen Ausführung werden Ausdrücke eines Quelltextes durch symbolische Repräsentanten ersetzt und Pfadanalysen auf Basis der resultierenden, formalen Ausdrücke durchgeführt. Somit wird ein Programm ohne konkrete Eingabewerte ausgeführt und diese dadurch berechnet. Derartige Ansätze eignen sich besonders bei Programmen imperativer Programmiersprachen, wie bspw. C. Prominente Werkzeuge zur Abdeckungsanalyse sind CREST [BS08] und KLEE [CDE08]. Beide Werkzeuge können für Programme, die auf der Programmiersprache C basieren, eingesetzt werden. KLEE nutzt die symbolische Ausführung und Constraint Solving Techniken zur Eingabedatenermittlung. Dabei wird zunächst der zugrundeliegende C-Quelltext instrumentiert (LLVM) und anschließend der symbolischen Umgebung (KLEE) übergeben. Der Constraint Solver (STP) ermittelt, zusammen mit einigen Optimierungen (Constraint Elimination, Caching, Environment Modeling), die Eingabedaten, mit denen alle erreichbaren Pfade durch den C-Quelltext abgedeckt werden können. CREST ist ein Werkzeug, das Techniken zur symbolischen und konkreten Ausführung anwendet. Dabei wird der C-Quelltext ebenfalls instrumentiert (CIL) und die extrahierten symbolischen Constraints mit dem SMT Solver Yices gelöst.

IEC 61131-3 basierte Applikationen können mit diesen Werkzeugen nicht unmittelbar analysiert werden. Unter gewissen Voraussetzungen können die Konstrukte von Strukturierten Text Programmen auf C-Programme abgebildet werden. Insbesondere aufgrund der Performance der Werkzeuge für symbolische Ausführung stellt diese als

einen vielversprechenden Kandidaten für die Anwendung bei IEC 61131-3 basierten Programmen dar.

Bewertung: Symbolic Execution Werkzeuge sind performante Werkzeuge für Abdeckungsanalysen, im Speziellen CREST und KLEE für C-Programme. Durch eine geeignete Transformation von IEC 61131-3 Strukturierten Text Bausteinen kann auch die zyklische Ausführungslogik nachgebildet werden, so dass Abdeckungsanalysen umgesetzt werden können.

3.4.3 Hybride Testverfahren

Hybride Verfahren kombinieren die Vorteile der strukturellen und funktionalen Testmethoden zur Generierung von aussagekräftigen Testfällen.

3.4.3.1 Ansätze hybrider Testverfahren

In [MW10] werden zufallsgesteuerte Testgenerierung, modellbasiertes Testen und Constraint Solving kombiniert, um effektive Testfälle abzuleiten. Auf Basis einer Anforderungsspezifikation werden Testdaten unter Zuhilfenahme der CECIL Methode (Cause-Effect Coverage Incorporating Linear boundaries), einer Kombination aus Ursache Wirkungsanalyse, Äquivalenzklassenbildung und Grenzwertanalyse generiert. Zusammen mit dem durch den Benutzer spezifizierten Arbeitsablauf werden die Testfälle in Bezug auf Abdeckungskriterien (bspw. Anweisungsabdeckung) generiert. Der sog. hybride Algorithmus ordnet Blöcke zufällig an, so dass diese gültige Testabläufe repräsentieren. Constraint Solving Techniken und Heuristiken optimieren den Erfüllungsgrad der geforderten Abdeckung, indem Pfade zu den Endpunkten als lokale Optima gesucht werden. Variationsparameter des hybriden Algorithmus umfassen Testfallparameter (bspw. Zeitlimit), Abdeckungsparameter (bspw. Abdeckungsgrad) und hybride Parameter (bspw. Suchtiefe) als Endkriterien für die Optimierung. Die Evaluation erfolgt an dem Open Source Projekt FileZilla.

Bewertung: Der hybride Ansatz verwendet zur Testdatengenerierung eine Anforderungsspezifikation, aus der zusätzlich Arbeitsabläufe manuell beschrieben werden. Der Testfallgenerierungsprozess wird durch Abdeckungskriterien skaliert. Eine zielgerichtete Reduktion der Testdaten ist nicht vorgesehen. Der Ansatz eignet sich nicht für IEC 61131-3 basierte Applikationen, da die zyklische Ausführungslogik während der Generierungsphase nicht berücksichtigt wird.

Das Projekt logi.DIAG adressiert zwei Aspekte des Softwaretests [log09], nämlich den statischen sowie den dynamischen Test. Die statische Codeanalyse umfasst Verfahren, die die Qualität der Software gemäß vorgegebener Konventionen überprüft. Dazu zählen neben der reinen Prüfung auf Einhaltung der Syntax, Abweichungen der Programmierkonvention sowie Kontrollfluss- und Datenflussanomalien. Darüber können die folgenden Fehler identifiziert werden: offene Verbindungen bei FBs, mehrfach beschriebene Variablen, toter Code, nicht benutzte Variablen und die Einhaltung von Namenskonventionen. Durch den schlüsselwortbasierten funktionalen Test wird das durch IEC 61131-3 kompatiblen Code beeinflusste Systemverhalten verifiziert.

Die Spezifikation der Testfälle erfolgt in Microsoft Excel. Ein sog. Test-Controller interpretiert diese Testfälle und bringt diese zur Ausführung.

Bewertung: Die statische Analyseverfahren, die in dem Projekt logi.DIAG erarbeitet wurden, dienen der allgemeinen Analyse des Quelltextes und nicht der Auswertung der Eingabedaten zur Testeingabedatenreduktion. Die dynamischen Tests erfordern eine vorgelagerte Spezifikation und werden nicht automatisch aus dem Quelltext abgeleitet.

3.4.4 Diversifizierende Testverfahren

Neben den funktionalen, strukturellen und hybriden Testverfahren existieren als weitere Ausprägung diversifizierende Testverfahren. Als die wesentlichen Hauptvertreten sind der Mutationstest und der evolutionäre Test anzuführen. Der Mutationstest, vorgeschlagen von DeMillo et al. [Dem78], zählt zu den fehlerorientierten Testmethoden und gehört zur Klasse der strukturbasierten (White Box) Testverfahren. Das Verfahren dient in erster Linie zur qualitativen Bewertung der Testfälle, indem eine Softwareversion (Mutant) durch sog. Mutationstransformationen modifiziert und im nachgelagerten Test geprüft wird, ob die Testfälle diese Veränderung identifizieren kann. Die Aussagekraft und Güte der Testfallverfeinerung hängt zu einem hohen Maß von der Wahl der Transformationsvorschrift ab. Formale Aspekte zum mutationsbasierten Test sind in [AH09] beschrieben. Eine umfassende Analyse zum Mutationstest wird in [JH11] dargestellt. Der evolutionäre Test wird für die Erzeugung von Testdaten eingesetzt. Das iterative Verfahren basiert auf der Evolutionstheorie (Darwinismus), so dass initial existierende Individuen gemäß einer Fitnessfunktion bewertet und die jeweils fittesten Individuen selektiert werden. Die Generierung von Testdaten wird somit zu einem Optimierungsproblem. Dieses Testverfahren kann für zahlreiche Testarten, strukturell als auch funktional eingesetzt werden.

3.4.4.1 Mutationstest

Eine systematische, modellbasierte Testdatenermittlung aus Signalflussplänen für zeitbehaftete Automatisierungssysteme wird von Lindner vorgestellt [Lin08]. Die zur Systemmodellierung und Spezifikation von Automatisierungssoftware häufig eingesetzten Signalflussdiagramme bilden die Basis der Testdatengenerierung durch eine systematische Auswertung der Modellinformationen. Das Testdatenermittlungsproblem wird gemäß der Testabdeckungskriterien als Constraint Problem (CSP) formuliert und Testdaten mit Hilfe von Constraint Lösungsverfahren ermittelt. Eine mutationsbasierte Formulierung des CSP als methodische Erweiterung bildet die Grundlage für ausdrucksstarke Testdaten. Der Ansatz wird an einem Kfz Karosserieelektroniksystem evaluiert. Die dabei erhaltenen Testdaten wurden mit denen verglichen, die aus gängigen Testverfahren abgeleitet wurden.

Bewertung: Die aus einer Spezifikation (Signalflussdiagrammen) automatisch abgeleiteten Testdaten erfüllen die Anforderung der hohen Aussagekraft und des geringen Aufwands in der Erstellung. Der präsentierte Ansatz stellt kein Konzept für fehlerbehaftete Komponenten dar und ist nicht direkt auf IEC 61131-3 Quelltext anwendbar.

Auf der Basis endlicher Zustandsautomaten als Systemspezifikation wird in [LI07] ein Ansatz zur Testdatengenerierung präsentiert. Es existieren zahlreiche Ansätze des Softwaretests auf Basis endlicher Zustandsmaschinen [Cho78], [CTF01].Durch den Einsatz genetischer Algorithmen, eine spezielle Klasse der evolutionären Algorithmen, werden Testeingabedaten in Bezug auf gegebene Anforderungen generiert. In dem zweigeteilten Ansatz werden zunächst Pfade des Zustandsautomaten in Bezug auf ein angegebenes Abdeckungskriterium (bspw. Transition) generiert. Die dadurch abgeleiteten Testsequenzen dienen als Eingabe für den zweiten Schritt. Für jede Sequenz werden durch den genetischen Algorithmus Eingabedaten gesucht, so dass Methoden in den Sequenzen ausgelöst werden. Die zur Berechnung notwendige Fitnessfunktion wird direkt abgeleitet und besteht aus zwei Komponenten: Abstandsmaß der Chromosomen zum Zielpfad und Erfüllungsgrad Vorbedingung. Der Vorschlag wird an zwei Java Klassen empirisch evaluiert. Hierarchische und nebenläufige Zustandsautomaten werden bislang nicht unterstützt.

Bewertung: Der präsentierte Ansatz verwendet endliche Automaten als Spezifikation und reduziert durch den Einsatz genetischer Algorithmen die Anzahl der Testeingabedaten. Die spezifischen Aspekte der zyklischen Ausführungslogik IEC 61131-3 basierter Applikationen wird in diesem Ansatz ebenso wenig adressiert wie die Berücksichtigung von Fehlersituationen.

Der Testfallgenerierungsansatz im Beitrag [PM10] vereint die dynamische symbolische Ausführung zusammen mit Transformationsvorschriften auf dem Ausgangsquelltext zur Automatisierung und Reduktion der generierten Testfälle. Die Autoren verknüpfen die dynamische symbolische (concolic[1]) Ausführung mit dem mutationsbasierten Ansatz. Es werden relationale und algebraische Ausdrücke ersetzt durch deren Umkehrung, was zu einer Komplexitätsreduktion in der Berechnung der Eingangswerte führt. Das sog. *Program under Test* wird in ein Metaprogramm transformiert, in dem Operandenpaare durch schematische Funktionen ersetzt werden, die das anschließende Mutationsverhalten repräsentieren. Die statische Struktur und der dazugehörige Kontrollflussgraph dienen dem Testgenerierungsverfahren als Eingangsdatensatz. Alle generierten Testfälle werden zur Ausführung gebracht und deren Verhalten (durchlaufene Pfade) analysiert. In mehreren Iterationszyklen entsteht so ein Mutationswert, der nach Studien von Andrews et al. [1] vermutlich der Fehlererkennungsrate entspricht. Der Ansatz wurde für Java Anwendungen entworfen und umgesetzt und an Beispielprogrammen (40-500 Zeilen Code) evaluiert. Die Anzahl der erzeugten Testfälle liegt mit zwischen 90 und 8927 relativ hoch.

Bewertung: Aus einem vorliegenden Quelltext extrahiert das mutationsbasierte concolic Verfahren Testfälle. Die Anzahl der generierten Testfälle fallen, gemessen an der Grundkomplexität der Programme, mit Faktor 2 bis 20, in Relation zur Anzahl der Zeilen Code, relativ hoch aus. Es wird u.a. keine Datenabhängigkeit zwischen den Zweigen mit berücksichtigt, so dass der Ansatz nicht für zyklische Applikation (IEC 61131-3) anwendbar ist.

1 Es handelt sich hierbei um einen Neologismus. Das Wort ist eine Mischform aus concrete und symbolic.

Eine Methode zur fehlerbasierten Testfallgenerierung aus einer OCL Spezifikation über die Vor- und Nachbedingungen wird in [APS05] vorgeschlagen. Dabei werden Fehler als eine mutierende Spezifikation modelliert. Der Testfallgenerierungsansatz erfolgt dabei auf einer automatischen Äquivalenzklassenbildung der Eingabedaten, in dem die Ein- Ausgabedatenrelation als Constraint Satisfaction Problem (CSP) formuliert und gelöst wird. Das Verfahren generiert für das angegebene Dreieckbeispiel die relevante Anzahl an Testdaten bzw. Testfällen.

Bewertung: Der allgemein formulierte Ansatz eignet sich für alle Systeme, die über OCL Spezifikationen beschreibbar sind. Für Applikationen mit zyklischer Ausführungslogik, wie IEC 61131-3 basierte Software können die zur Erreichbarkeit bestimmter Anweisung notwendigen Eingabedaten nicht generiert werden, da das CSP Verfahren nur für die Relation zwischen den Vor- und Nachbedingungen greift.

3.4.4.2 Evolutionärer Test

Ein evolutionärer Black Box Test für eingebettete Systeme (Steuergeräte im Automobil) wird in [KWW09] präsentiert. Das Werkzeug MESSINA ist ein Testwerkzeug zur hard- und softwareunabhängigen Spezifikation von Testsequenzen. Dazu werden die Notationsformen UML, Java oder TPT eingesetzt. In Kombination mit dem modular-HIL System integriert sich das evolutionäre Test Framework zum Gesamtsystem. Die in MESSINA generierten Testfälle werden aufgrund von erzeugten Individuen gegen ein Testobjekt zur Ausführung gebracht und deren relevanten Fitnesswerte an das Framework zurückgegeben. Die Fitnessfunktion wird für jeden abgearbeiteten Testfall neu bewertet, indem ein generischer Testfall ausgeführt wird. Für zeitliche Eigenschaften eingebetteter Systeme können Fitnessfunktionen weitestgehend generisch erstellt werden, wohingegen sie für funktionale Aspekte manuell entworfen werden müssen. Das Gesamtsystem wurde an einem ABS System verifiziert.

Bewertung: Der Ansatz beschreibt eine integrale Erweiterung eines bestehenden Systems zum Test von Steuergeräten im Automobil. Aus dem Artikel geht nicht vollständig hervor, woraus und auf Basis welcher Daten die evolutionäre Entwicklung der Tests durchgeführt wird. In dem Beispiel wurden besonders jene Testdaten gefunden, die an den Systemgrenzen anliegen. Für zyklische Applikationen ist der Ansatz nicht geeignet.

Vudatha et al. beschreiben ein Testfallgenerierungsverfahren mit evolutionären Algorithmen auf Basis der Benutzeranforderungsspezifikation [VJND11]. Zur Ermittlung der Eingabedaten werden als Zielfunktion die erwarteten Ausgabedaten zu Grunde gelegt. Der Algorithmus wird solange ausgeführt, bis die Population alle Ausgabedaten abdeckt. Das Verfahren wird an einem Temperaturüberwachungs- und steuerungssystem von Nuklearreaktoren beschrieben.

Bewertung: Aus dem Beitrag geht die Allgemeingültigkeit des Ansatzes durch den starken Argumentationsbezug auf das Applikationsbeispiel nicht hervor. Es bleibt ungeklärt, für welche Zielsysteme das Verfahren geeignet ist. IEC 61131-3 spezifische Aspekte der Generierung werden nicht explizit adressiert.

3.4.5 Gesamtbewertung Testverfahren

Die Testverfahren in diesem Kapitel wurden gemäß ihren charakteristischen Eigenschaften funktional, strukturell, hybrid und diversifizierend kategorisiert präsentiert. Eine Bewertung der jeweiligen Ansätze erfolgte auf Basis der in Kapitel 3.1.2 eingeführten Kriterien. Tabelle 3.3 zeigt die bewerteten Ansätze in einer Gesamtübersicht. In der Auflistung wird deutlich, dass im Maschinen- und Anlagenbau bislang kein Ansatz zur expliziten Generierung und Reduktion von Testdaten für IEC 61131-3 basierte (zyklische) Applikationen existiert. Neben Verfahren zur Generierung von Testfällen aus Spezifikationen, wie bspw. UML Zustandsdiagrammen, existieren in der Domäne Ansätze zur formalen Verifikation [SF09], [TSF11], [MF01], [LBYF05], [MWD05], [VH05] der Testfallbeschreibung [KCJ11], [KTVH12] und dem Testmanagement [SEK09].

In Domänen wie Automobil, Luft- und Raumfahrt und der Anwendungsentwicklung existieren zahlreiche Ansätze zur Test- bzw. Testdatengenerierung aus Anforderungen [Pfa10], [CR04], [Ott08] aus Spezifikationen [AJ12], [Bro10], [KHC99] sowie durch eine Strukturanalyse des Quelltextes [MB03], [ABLN06], [GC09], [AML11], [JH01] von Hochsprachen. Jedoch werden bei keinem dieser Ansätze Systemkomponentenfehler zur expliziten Generierung und Erhöhung der Aussagekraft berücksichtigt. Die dabei erarbeiteten Verfahren betrachten ereignisbasierte Systeme und sind aufgrund der durch die zyklische Ausführungslogik fehlenden Berücksichtigung der Aspekte nicht für IEC 61131-3 basierte Software anwendbar. Es existiert bislang kein Ansatz zur Testdatengenerierung aus Quelltextstrukturanalysen für zyklische IEC 61131-3 basierte Applikationen.

3.5 Zusammenfassung

In diesem Kapitel erfolgte eine Vorstellung und Bewertung einer Vielzahl existierender Ansätze. Auf Basis der Problemanalyse (Kapitel 2) und dem Stand der Technik zeigt sich weiterhin ein bestehender Handlungsbedarf. Dabei wurden Teilaspekte vereinzelt bereits realisiert, jedoch existiert kein ganzheitlicher Ansatz zur fehlerzentrierten Testdatengenerierung für IEC 61131-3 basierte Applikationen. Insbesondere sind dafür die folgenden Punkte bislang nicht berücksichtigt:

- *Integration von Fehlern (**/S4**)* Im Maschinen- und Anlagenbau existiert bislang kein Verfahren zur Integration von Fehlern in das Gesamtsystemmodell.
- *Zyklischen Ausführungslogik (**/T4**) und Quelltext (**/T5**)* Existierende Testdatengenerierungsverfahren leiten die notwendigen Eingabedaten häufig aus Spezifikationen aus und nicht aus dem vorhandenen Quelltext, da diese aufgrund der zyklischen Ausführungslogik mit den existierenden Analyseverfahren nicht ausgewertet werden kann.
- *Zyklischen Ausführungslogik (**/T4**) und Quelltext (**/T5**)* Existierende Testdatengenerierungsverfahren leiten die notwendigen Eingabedaten häufig aus Spezifikationen aus und nicht aus dem vorhandenen Quelltext, da diese aufgrund der zyklischen Ausführungslogik mit den existierenden Analyseverfahren nicht ausgewertet werden kann.

- *Testdatenreduktion (**/T2**) und Aussagekraft (**/T7**)* Durch den bislang fehlenden Bezug zu Fehlern in mechatronischen Systemen, ist die Aussagekraft automatisch generierter Testdaten häufig nicht gegeben, da kein explizites Testziel (bspw. Fehlerbezug) eingesetzt wird. Eine gezielte Testdatenreduktion zur praktischen Durchführbarkeit ist häufig wegen des hohen Parameterraums nicht gegeben.

Tab. 3.3: Bewertungsübersicht der Testverfahren gemäß den Kriterien aus Kapitel 3.1.2

Verfahren / Quelle	Fehlerbezogen	Testdatenreduktion	Ausführungsart	Zyklische Ausführungslogik	Strukturorientiert	Geringer Aufwand	Aussagekraft der Testdaten
Funktional / [HKVH11]	○	○	●	●	○	●	○
Mutation / [Lin08]	○	◑	●	○	○	●	●
Funktional / [KHD08]	○	○	●	○	○	●	○
Funktional / [WS08]	○	●	●	○	○	◑	●
Hybrid / [log09]	○	○	●	●	○	◑	◑
Funktional / [SP10]	○	○	●	○	○	●	○
Formal / [KWV10]	○	○	●	●	●	●	○
Mutation / [CTF01]	○	◑	◑	○	○	◑	◑
Funktional / [Kel09]	○	◑	●	○	○	○	◑
Mutation / [LI07]	○	◑	●	○	○	◑	◑
Strukturell / [GBR98]	◑	●	●	○	●	●	◑
Strukturell / [AERP04]	○	○	●	●	●	●	●
Strukturell / [MK09]	○	●	●	○	●	●	◑
Funktional / [ZuM15]	●	○	●	●	◑	○	○
Mutation / [PM10]	○	●	●	○	●	●	◑
Hybrid / [MW10]	○	○	●	○	◑	◑	●
Evolutionär / [KWW09]	○	◑	◑	○	◑	●	◑
Evolutionär / [VJND11]	○	●	◑	○	◑	●	●
Mutation / [APS05]	◑	●	●	○	○	●	●
Strukturell / [JWAW10]	○	◑	●	○	◑	◑	◑
Strukturell / [NWM02]	●	●	●	○	●	○	◑

Legende: ●: Vollständig erfüllt ◑: Teilweise erfüllt / nicht feststellbar ○: Nicht erfüllt

Im nächsten Kapitel wird ein Konzept präsentiert, das die geforderten Anforderungen an eine fehlerzentrierte Testdatengenerierung für zyklische IEC 61131-3 Applikationen adressiert und die vorhandene Lücke schließt.

KAPITEL 4

Konzept zur automatisierten Testdatengenerierung fehlerbehafteter Systeme

In diesem Kapitel wird ein Konzept zur Fehlermodellierung und darauf aufbauenden Testdatengenerierung für Softwarebausteine mit zyklischer Ausführungslogik vorgestellt. Dazu wird zunächst eine Beschreibungsmethodik und Integration für Fehlermodelle in das technische Gesamtmodell erarbeitet. Der Formalismus zur Testdatengenerierung nutzt die Informationen des Fehlermodells zur zielgerichteten Ableitung.

Inhaltsverzeichnis

Software- und Systemtests sind die mit Abstand wichtigsten Maßnahmen zum Nachweis bzw. der Absicherung geforderter Qualitätskriterien [PJR09]. Dabei ist besonders eine spezifische Teststrategie mit den geeigneten Testverfahren essentiell. Zur Erhöhung der Fehleridentifikationsrate werden in Vorgehensmodellen, wie bspw. dem 3-Ebenen-Vorgehensmodell [Ben05], Tests auf allen Entwicklungsebenen zur Prüfung der in der jeweils korrespondierenden Phase geforderten Aspekte integriert. Die Erörterung in Kapitel 2.2.1 zeigt, dass derartige Maßnahmen im Maschinen- und Anlagenbau derzeit nicht umfangreich genug umgesetzt werden können, da u.a. kaum Methoden und Werkzeuge zur domänenspezifischen Anwendung zur Verfügung stehen [KTVH12].

Die Wahl der zur Testdurchführung notwendigen Kriterien und Testeingabedaten stellt dabei einen entscheidenden Faktor bei der Bewertung der praktischen Durchführbarkeit und der gezielten Absicherungsbewertung dar. Bei der Inbetriebnahme und während des Betriebs von Maschinen bzw. Anlagen sorgen besonders Fehlersituationen, die im schlechtesten Fall zu einer Gefährdung von Mensch und/oder Maschine führen können, für einen erhöhten Absicherungsaufwand. Die Analyse einschlägiger Literatur in Kapitel 3 zeigt, dass bislang keine zufriedenstellende Lösung existiert, die die in Kapitel 2 erhobenen Anforderungen erfüllt.

Dieses Kapitel stellt eine Lösung zur fehlerorientierten Testdatengenerierung für zyklische SPS Steuerungsapplikationen vor. Im Folgenden werden in 4.1 zunächst ein Gesamtüberblick über das Konzept und die notwendigen Handlungspunkte aufgeführt. In 4.2 erfolgt die Einführung in die graphische Modellierungssprache zur Beschreibung des Systemmodells und den damit verbundenen Aufbau sowie die Integration eines Fehlermodells. Die formalen Konzepte zur automatischen Generierung von Testdaten für zyklische Softwarebausteine wird in 4.4 beschrieben.

4.1 Grundidee und Konzeptüberblick

Bei der Entwicklung technischer Systeme basieren die Auslegung, Designentscheidungen und Absicherungsmaßnahmen auf den an das System gestellten funktionalen und nichtfunktionalen Anforderungen. Aufgrund steigender Notwendigkeit der Traceability werden Anforderungen horizontal im Sinne der Verifikation und Validierung wie auch vertikal bis zum elementaren Umsetzungsbestandteil gefordert.

Dieses Vorgehen reicht jedoch meist nicht aus, da Systeme letztendlich einerseits nicht identifizierte Anforderungen nicht enthalten, geforderte Anforderungen nur mangelhaft umgesetzt sind und andererseits aufgrund mangelnder Verfahren während der Verifikation nicht aufgedeckt werden können. Dies führt zu einer Abweichung des erwarteten Systemverhaltens, welches in gewissen Situationen als Fehler interpretiert wird [KVH11].

Die Grundidee der Fehlerbetrachtung dieser Arbeit entspricht der expliziten Berücksichtigung von Fehlern technischer Komponenten (Sensor, Aktor) von Maschinen und Anlagen zur Selektion des Testziels der Testdatengenerierung. Reine Softwareimplementierungsfehler, wie bspw. Division durch Null sind somit nicht Gegenstand des Fehlermodells. Das Fehlermodell dient der Selektion eines potentiellen Fehlers einer technischen Komponente, wofür Eingabedaten der Steuerungsapplikationen derart

generiert werden müssen, so dass bei einer Testausführung die potentiell fehlerhafte technische Komponente angesprochen wird. Durch die Beobachtung und Interpretation des Systemverhaltens kann eine Testbewertung erfolgen. Das Gesamtverhalten kann aufgrund der folgenden Situationen in der SPS Steuerungssoftware erwartungskonform oder fehlerhaft sein:

- Die Routine zur Behandlung des Fehlers der technischen Komponente ist in der Steuerungssoftware nicht korrekt bzw. unvollständig umgesetzt.
- Die Steuerungssoftware besitzt keinen expliziten (implementierten) Fehler. Jedoch kann der Fehler der technischen Komponente durch die vorliegende Implementierung nicht kompensiert werden, was u.U. zu einem ungewollten Verhalten des Systems führt.
- Implementierungsfehler der Steuerungssoftware, wie bspw. Fehler in der Fallunterscheidung oder nicht erreichbarer Quelltext, können ebenfalls die Ursache für fehlerhaftes Gesamtverhalten darstellen.

Da auf ein System – unabhängig von den daran gestellten Anforderungen – die physikalischen Gesetzmäßigkeiten der Umwelt einwirken und das Verhalten durch die beabsichtigte und unbeabsichtigte Umsetzung bestimmt wird, bedarf es eines Verfahrens zur Generierung von Stimuli, die auf der tatsächlichen Implementierung basieren. Somit kann das Systemverhalten für ausgewählte fehlerbehaftete Situationen gezielt verifiziert werden. Das Gesamtkonzept wurde in [KS16], [KVH11] und [TKVH12] publiziert.

Zur Einordnung der Grundidee und des Konzepts sind in Abbildung 4.1 die Abläufe und Zusammenhänge dargestellt. Die Systementwicklung lässt sich generell in drei Phasen unterteilen. Während der Phase der Software- und Modellerstellung müssen alle relevanten Artefakte entwickelt werden. Die SPS Steuerungssoftware stellt die wesentliche Funktionserfüllung eines Systems dar. Die Steuerungssoftware wird bei der Testdatengenerierung für die Pfadanalyse benötigt. Bei der Elektroplanerstellung müssen Signalverlauf, Not-Aus-Ketten sowie der Schnittstellenvertrag zwischen Steuerung und realem System, die sog. I/O Liste, festgelegt werden. Die I/O Liste dient dazu, den Bezug von Komponenten des technischen Systems zu deren Repräsentation in der Steuerungssoftware herzustellen. Dies wird sowohl bei der Testdatengenerierung benötigt als auch für die Kopplung des Steuerungssystems an eine Simulation des realen Systems (Phase der Testausführung). Das System- und Fehlermodell ist Gegenstand dieser Arbeit und wird in Abschnitt 4.2 im Detail beschrieben. Es stellt den Startpunkt der Testdatengenerierung dar, worüber der Fehler und somit die defekte Komponente selektiert wird. Ein weiterer Bestandteil des Modells ist die sog. Sensor-Aktor-Relation, die in Abschnitt 4.4.4 eingeführt wird. Mit Hilfe dieser Relation können funktionale Querbezüge, die aufgrund des technischen Prozesses eine gemeinsame Funktion existieren, bei der Testdatengenerierung berücksichtigt werden. Dadurch können Testdaten für den gleichen Fehler mit unterschiedlicher Ursache generiert werden.

Die Testdatengenerierung benötigt die Auswahl des Fehlers durch den Tester, für den die Testeingabedaten generiert werden sollen. Über das System- und Fehlermodell kann

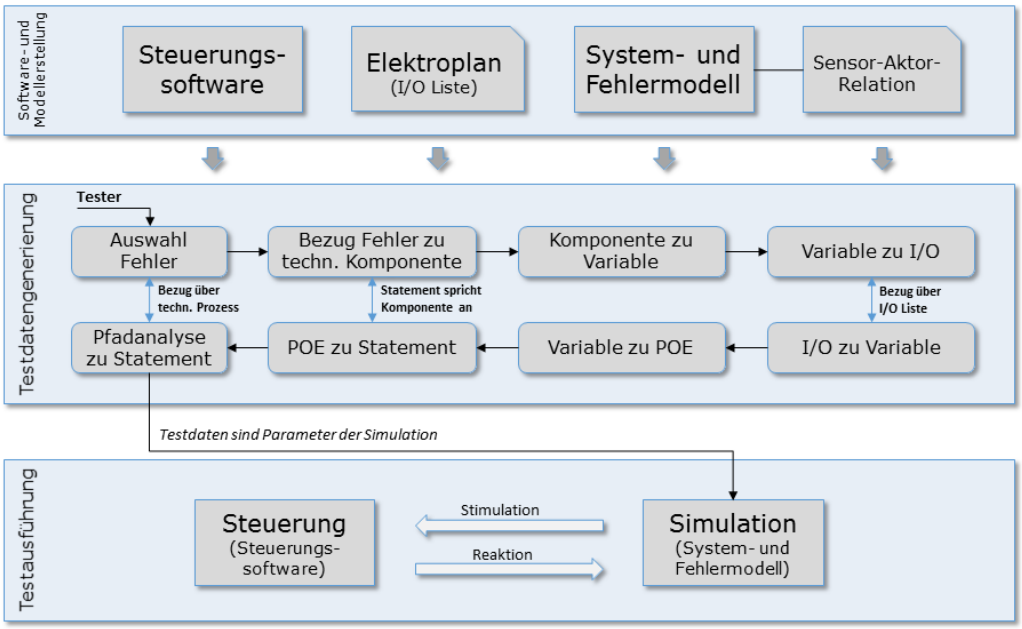


Abb. 4.1: Konzeptüberblick – Ablauf und Zusammenhänge

der Bezug zu der technischen Komponente hergestellt und die weiteren Abstraktionen bis zu den I/O Registern aufgelöst werden. Durch die Auswertung der I/O Liste erfolgt der Übergang in die Steuerungssoftware, so dass das Statement identifiziert werden kann, welches die betroffene technische Komponente anspricht. Die nun auf die Steuerungssoftware angewendete Pfadanalyse ermittelt die Eingabedaten derart, dass bei anschließender Stimulation der Steuerungssoftware das identifizierte Statement erreicht wird. Das Konzept der Testdatengenerierung wird in den Abschnitten 4.3, 4.4 und 4.5 erläutert. Für das Verfahren werden die folgenden Voraussetzungen festgelegt:

- **Testorakel** Die Soll-Verhalten des System muss manuell (Tester) festgelegt werden, da das beschriebene Verfahren auf einer Quelltextanalyse und nicht auf einer Spezifikation, welche das Soll-Verhalten beschreibt, basiert.
- **Steuerungssoftware** Der Quelltext der Steuerungssoftware liegt in Strukturiertem Text (ST) vor. Die Berücksichtigung weiterer Sprachen wird in Abschnitt 6.4 beschrieben.
- **Kommunikation** Es existiert eine einkanalige Kommunikation zu den Sensoren bzw. Aktoren des technischen Systems.
- **Simulation** Es ist für die Steuerungssoftware transparent, ob diese gegen das reale System oder eine Simulation ausgeführt wird.
- **Test** Der Tester selektiert den gewünschten Fehler auf Basis des Testziels. Die durch die Testdatengenerierung ermittelten Eingabedaten müssen, zusammen mit dem ausgewählten Defekt, im Simulationssystem durch den Tester gesetzt werden.

Die ermittelten Testeingabedaten dienen als Parameter der Simulation, da sich diese in Interaktion mit der Steuerungssoftware befindet. Diese sind erforderlich, um die Steuerungssoftware so zu stimulieren, dass die als defekt markierte technische Komponente angesprochen wird. Während der Testausführung kann das Verhalten des Gesamtsystems durch den Tester beobachtet und mit dem erwarteten Verhalten verglichen werden.

4.1.1 Ansatz zur automatisierten Testdatengenerierung

In Abbildung 4.2 ist die Architektur des Gesamtkonzepts zur fehlerorientierten Testdatengenerierung dargestellt. Die in rot hervorgehobenen Bestandteile sind konkreter Gegenstand dieses Kapitels. Das Fehlermodell für technische Komponenten wird in das Systemmodell integriert und dient zur Auswahl der Testdatengenerierung für die gekoppelte IEC 61131-3 Steuerungsapplikation. Das Konzept sieht eine bidirektionale Kopplung der Ein-/Ausgaben (I/O) der Steuerung zwischen der Applikation und dem Systemmodell, das bspw. in einer Simulationsumgebung eingebettet sein kann, vor. Dadurch ergibt sich eine eineindeutige Relation zwischen technischen Komponenten (Sensoren, Aktoren) und der Software-internen Repräsentation. In Softwarebausteinen der Applikationen werden, abhängig vom Funktionsumfang, Werte der technischen Komponenten (Sensoren) eingelesen und unter Berücksichtigung des internen Zustandes der Steuerungssoftware neu berechnete Werte an technische Komponenten (Aktoren) geschrieben. Die Testdatengenerierung basiert auf einer formalen Pfadanalyse der gegebenen IEC 61131-3 Steuerungsapplikation. Die Art der Erstellung ist unabhängig für den automatischen Prozess der Testdatengenerierung, d.h. diese kann entweder vollständig manuell erstellt oder aus einer vorangegangenen Modellbildung in ein ablauffähiges Steuerungsprogramm transformiert werden [WVH11], [WRK10], [WSVH08]. In Kapitel 4.1.2 werden weitergehende Aspekte zur Steuerungssoftware eingeführt.

Bei der Systementwicklung beanspruchen Fehlersituationen, die im Gesamtverbund, insbesondere während des Betriebs auftreten können, einen Großteil des gesamten Steuerungssoftwareumfangs [Mon01]. Bei der Entwicklung moderner Maschinen und Anlagen steht jedoch meist die primäre Anlagenfunktion im Vordergrund (Geradeauslauf), so dass der Fokus beim Abnahmetest darauf gerichtet wird. Die in der Steuerungssoftware hinterlegten Fehlerbehandlungsroutinen, welche der Stabilität und Zuverlässigkeit des Gesamtsystems dienen, werden jedoch aufgrund der hohen zeitlichen Anforderungen und der daraus resultierenden mangelnden Berücksichtigung des Tests nicht explizit getestet, siehe Kapitel 2.2.2. Meist liegen die typischen Fehlerfälle, bestehend aus deren Ursache, Zusammenhang, Auswirkungen und Eintrittswahrscheinlichkeiten in Form von mentalen Modellen vor, können jedoch aufgrund der informalen Existenz nicht nachweisbar in den Test integriert werden. Es wird demnach ein explizites Fehlermodell benötigt, das sich in das Gesamtsystemmodell integrieren lässt. So wird es möglich, relevante Aspekte zielgerichtet zu verifizieren. Eine Fehlerberücksichtigungen existiert im Test allgemein für UML Modelle [IAY01] und in der Anwendung im Automobilbereich [SV05].

Durch die I/O Kopplung der Steuerungsapplikation mit dem dazugehörigen System- und Fehlermodell können die Informationen über potentielle Fehler, die durch die

Steuerungsapplikation erkannt werden und diesen entgegenwirkt, der Testdatengenerierung zugrunde gelegt werden. Die formale Pfadanalyse des Quelltextes der Steuerungsapplikation ermittelt zusammen mit den Fehlerinformationen und der I/O Abbildung die relevanten Eingabeparameter zur Stimulation des Testobjekts. Über das in Kapitel 4.4 vorgestellte Verfahren wird eine minimale Menge an repräsentativen Testeingabedaten automatisch generiert.

An alle Bestandteile der Architektur werden die Anforderungen aus Kapitel 2.4.2 gerichtet. In den folgenden Kapiteln 4.2 und 4.4 werden die entsprechenden Komponenten detailliert beschrieben und es wird auf deren Anforderungserfüllung Bezug genommen.

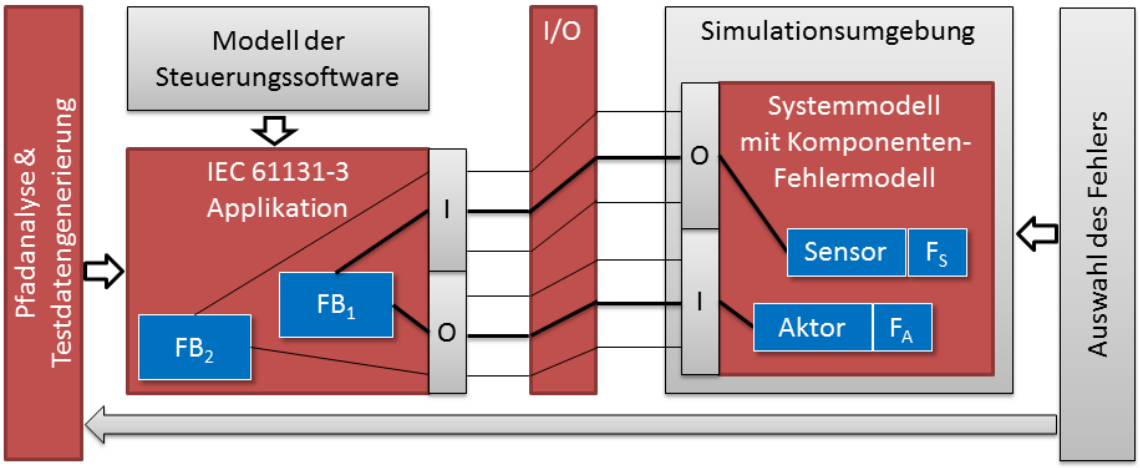


Abb. 4.2: Architektur des Gesamtkonzepts zur fehlerorientierten Testdatengenerierung

4.1.2 Struktur und Zusammensetzung der SPS Steuerungssoftware

Im Maschinen- und Anlagenbau werden Steuerungsprogramme abhängig von der jeweiligen Branche, dem Sonder- oder Serienmaschinenbau, etablierten Architekturen bzw. Frameworks und der vorhandenen Funktionsbibliothek entwickelt. Dabei reichen die Vorgehensweisen von einer manuellen Neuimplementierung einer bereits vorhandenen Funktionalität über einen copy-and-modify[1] Ansatz bis zur vollständig modellbasierten Entwicklung. Des Weiteren bestehen Abhängigkeiten zu anderen Disziplinen, so dass die Entwicklung (Strukturierung der Software, Bezeichnung von Ein-/Ausgaben) maßgeblich durch andere Disziplinen und den Anforderungen an das Gesamtsystem beeinflusst wird [HP09], [Hau06], [Sch06].

Ein SPS Steuerungssystem wird nach IEC 61131-3 unter dem Begriff *Configuration* zusammengefasst. Darin können mehrere Steuergeräte als sog. *Ressources* eingegliedert werden, auf der jeweils ein oder mehrere *Tasks* mit den individuellen Laufzeitparametern festgelegt werden. In den einzelnen Tasks werden Programme (SPS Applikationen)

1 Hierbei werden vorhandene Softwarefunktionen kopiert und an die neue Steuerungsplattform oder das neue Projekt angepasst.

zur Ausführung gebracht, die in den fünf verfügbaren Programmiersprachen realisiert werden können. Eine SPS Applikation wird aus Programmorganisationseinheiten (POE) konstruiert. Dabei werden im Wesentlichen drei Typen unterschieden. Die in Tabelle 4.1 beschriebenen POE-Typen besitzen die folgenden Eigenschaften:

- **Programm**: Diese Organisationseinheit stellt das Hauptprogramm, den Haupteinsprungpunkt der Steuerungsapplikation dar. Durch den statischen Charakter von IEC 61131 Programmen werden alle Variablen und deren Kopplung an die physikalischen Ein-/Ausgänge darin definiert.
- **Funktionsbaustein**: Ein Funktionsbaustein stellt aus Umsetzungs- und Analysesicht die komplexeste POE dar. Durch die Verwendung statischer Variablen (Gedächtnisfunktionalität) hängen im Vergleich zur Funktion die Ausgabewerte nicht alleine von der Belegung der Eingabeparameter ab, sondern auch vom internen Zustand des FBs, d.h. der Belegung der internen statischen Variablen. Bei einer erneuten Ausführung können demnach andere Pfade durch die Implementierung gewählt werden und ein abweichendes Ergebnis zur Folge haben.
- **Funktion**: Ein POE dieser Art stellt eine primitive Ein-/Ausgangsrelation zur Auslagerung häufig benötigter kombinierter Operationen dar. Funktionen besitzen selbst kein Gedächtnis, so dass deren Ausgangswerte allein von der Belegung der Eingabeparameter abhängen. Typische Beispiele hierfür sind Konvertierungsfunktionen zwischen Koordinatensystemen, Geschwindigkeiten oder Temperaturwerten.

Tab. 4.1: Beschreibung der IEC 61131-3 Programmorganisationseinheiten [JT09]

POE-Typ	**Schlüsselwort**	**Bedeutung**
Programm	PROGRAM	Hauptprogramm mit Zuordnung der SPS-Peripherie, globalen Variablen und Zugriffspfaden
Funktionsbaustein	FUNCTION_BLOCK	Baustein mit Ein- und Ausgangsparameter, ist der zur Programmierung hauptsächlich benutzte POE-Typ
Funktion	FUNCTION	Baustein mit Funktionswert, Ein- und Ausgangsparameter zur Erweiterung des SPS-Operationsvorrats

Neben dem einmalig vorhandenen Hauptprogramm können beliebig viele Funktionsbausteine und Funktionen zur Applikationsrealisierung erstellt werden. SPS Programme werden im Vergleich zu ereignisgesteuerten Systemen zyklisch (time-boxed) ausgeführt, was eine unmittelbare Auswirkung auf die algorithmische Umsetzung nach sich zieht. Die daraus resultierenden Besonderheiten für die Testdatengenerierung

werden in Kapitel 4.4 vollständig beschrieben. Zur Implementierung der POEs stehen in der IEC 61131-3 fünf Programmiersprachen zur Verfügung, welche ineinander überführbar sind. Funktional besteht kein Unterschied. Die Testdatengenerierung erfolgt unabhängig von der angewandten Entwicklungsmethodik und ist auf alle IEC 61131-3 kompatiblen Sprachen anwendbar.

4.2 Fehlerzentrierte Systemmodellierung

Für die Ableitung von Testdaten, die potentielle Fehlerszenarien eines Systems adressieren, bedarf es einer Systemmodellierung zur Integration von Fehlermodellen. Die daran gestellten Anforderungen und Bewertungskriterien wurden in den Kapiteln 2.4.2 und 3.1.1 erhoben und definiert.

In Kapitel 3.2 wurden Fehlertypen, Fehlerklassifikationen und Fehlerursachen eingeführt. Trotzdem dreigliedriger Unterscheidung von Fehlertypen, beschreibt keine dieser Typen, was unter einem Fehler zu verstehen ist. Gemäß der Motivation am Anfang des Kapitels beziehen sich Fehler auf technische Komponenten einer Maschine oder Anlage. Entsprechend wird für diese Arbeit das Fehlermodell nach Definition 8 eingeführt. Im Kontext der Arbeit entsprechen Fehler demnach einer Verhaltensausprägung technischer Komponenten einer Maschine oder Anlage, die bspw. aufgrund der physikalischen Gesetzmäßigkeiten eintreten können. Sofern ein geforderter Funktionsumfang durch die Applikation einer SPS ausgeführt wird, führt dies im Fehlerfall einer technischen Komponente zu einem beobachtbaren Verhalten. Ob es sich dabei um den geforderten Funktionsumfang handelt, ist von der Qualität (Stabilität, Fehlertoleranz etc.) der Steuerungssoftware abhängig. Sofern das Verhalten abweicht, bedarf es der Interpretation des Verhaltens, ob es sich um ein fehlerhaftes Verhalten des Gesamtsystems handelt. Diese Interpretation kann nur nach vorheriger Beobachtung des Systemverhaltens durch den Menschen[1] erfolgen. Die Auswirkungen eines Fehlers technischer Komponenten auf das Gesamtverhalten bzw. die Aktivierung des Fehlers im an die Testdatengenerierung anschließenden Test ist nicht Teil dieser Arbeit.

Definition 8 (*Fehlermodell*)

Ein Fehlermodell entspricht einer abstrahierten, bestimmten Verhaltensausprägung technischer Komponenten (Sensor, Aktor), deren angeforderte, beobachtbare Funktion bei Eintritt des Fehlers (error) zu einem ungewollten Gesamtverhalten (failure) führen kann. Die eigentliche Ursache (fault), die den Eintritt des Fehlers (error) nach sich zieht, ist dabei unerheblich.

Das Fehlermodell dient zur expliziten Modellierung und Integration von Verhaltensausprägungen technischer Komponenten und dient anschließend als Grundlage für die Auswahl des Testziels, d.h. der Generierung von Testeingabedaten der SPS

1 Eine Automatisierung oder explizite Modellierung ist i.a. nicht möglich, da das Verhalten häufig vorab nicht bekannt ist und die Klassifikation des Verhaltens, meist auch abhängig von der Personengruppe, unterschiedlich bewertet wird. Deshalb erfolgen derartige Bewertungen stets in einem Gremium, da dadurch eventuell notwendige Änderungen am System auf deren Auswirkungen, Notwendigkeit und Aufwand hin bewertet werden müssen.

Steuerungsapplikation, die durch Stimulation mit den generierten Daten, die potentiell fehlerhafte Komponente ansprechen und so das Systemverhalten beobachtet werden kann. Der Umfang und der Detaillierungsgrad bei der Modellierung skaliert mit der Erfahrung und den Absicherungsanforderungen des Maschinen- und Anlagenbauers.

Als Basis zur System- und Fehlermodellierung dient die SysML, die zahlreichen Ansätzen zugrunde gelegt wird [Thr10], [BSBF10], [SW09]. Die SysML bietet eine methoden- und werkzeugunabhängige Modellierungssprache mit graphischen Notationselementen (**/S3**) und eignet sich konzeptionell zur domänenspezifischen Verwendung im Maschinen- und Anlagenbau. Die Aspekte des in diesem Unterkapitel präsentierten Konzepts wurden in [TKVH12] publiziert.

4.2.1 Modellübersicht und Erweiterung des Sichtenkonzepts

Das Drei-Sichten-Konzept [LG99], [SW09] betrachtet die drei folgenden Aspekte:

- **Technischer Prozess**: „Gesamtheit von Vorgängen, bei denen physikalische Größen mit technischen Mitteln erfasst (Ergebnisgrößen) und beeinflusst werden (Einflussgrößen)“ [LG99]
- **Automatisierungssystem**: „Rechner- und Kommunikationssystem, in welchem ein Informationsprozess abläuft (Umformung, Verarbeitung und Transport von Information)“ [LG99]
- **Technisches System**: „Technisches Produkt oder eine technische Anlage, in dem ein technischer Prozess (Umformung, Verarbeitung und Transport von Materie oder Energie) abläuft“ [LG99]

In Abbildung 4.3 sind die drei Sichten auf eine Maschine bzw. Anlage schematisch dargestellt. Das Drei-Sichten-Konzept ermöglicht eine gemeinsame, integrierte Modellbildung, da es die unterschiedlichen Gewerke unterstützt und gleichzeitig eine Reduktion der Komplexität auf die jeweilige Sicht fördert. Abbildung 4.4 zeigt den Gesamtüberblick über die Bestandteile des hierarchischen SysML Modells eines Gesamtsystems als Erweiterung des drei Sichtenmodells [SW09]. Dabei liegt hier der Fokus auf der Modellierung des technischen Systems mit allen relevanten Komponenten (**/S2**). Das Modell des technischen Systems integriert sich gemäß dem oben beschriebenen Sichten-Konzept in das Gesamtmodell (**/S1**), das um Aspekte im Hinblick auf den fehlerorientierten Test durch den sog. Systemkontext und eine auf Wiederverwendung ausgelegte Fehlerbibliothek (**/S4**) erweitert wird. Die einzelnen Modellelemente zur strukturellen und verhaltensbasierten Beschreibung des zu entwickelnden technischen Systems, der Systemkontext und das Konzept der Fehlerbibliothek werden in den folgenden Unterkapiteln 4.2.1.1, 4.2.1.2 und 4.2.2 näher beschrieben.

4.2.1.1 Struktur- und Verhaltensmodellierung des technischen Systems

Die Modellierung der Systemstruktur erfolgt durch die anforderungsspezifische Beschreibung der Komponenten unter Einsatz der beiden SysML Diagramme Blockdefinitionsdiagramm (BDD) und internes Blockdiagramm (IBD). Das BDD dient

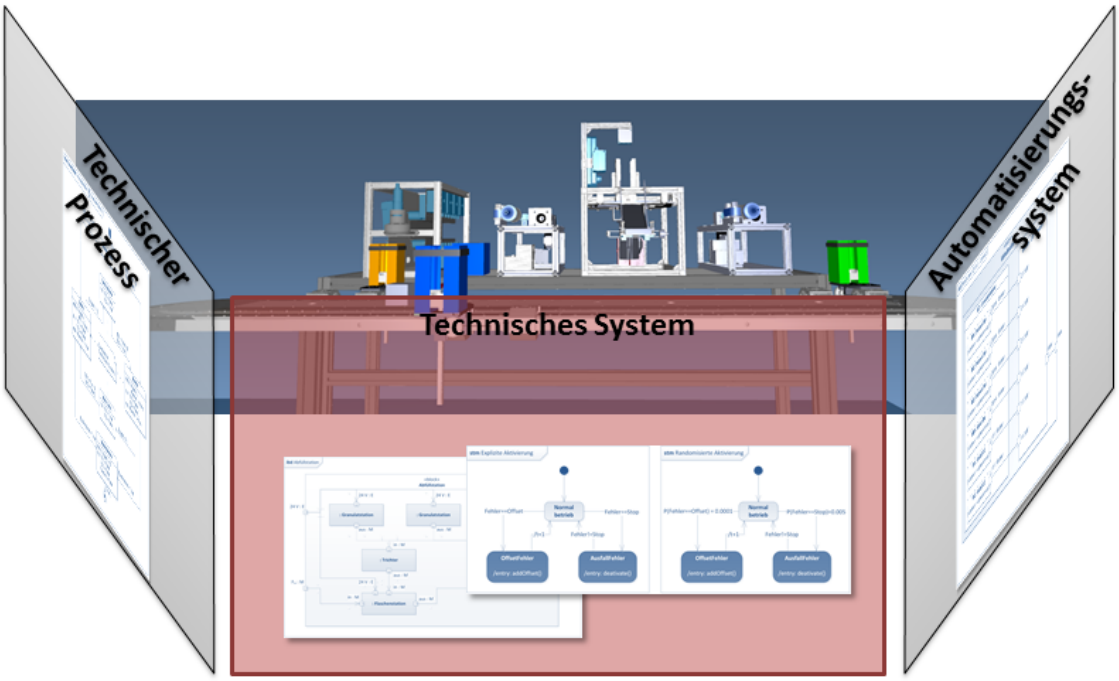


Abb. 4.3: Über Sichten repräsentiertes Prozessautomatisierungssystem

dabei der Black Box Darstellung der Komponenten und entspricht dem Grobkonzept. Die White Box Beschreibung, d.h. die intrinsischen Strukturen, Beziehungen und Flusszusammenhänge werden mit dem IBD als Feinkonzept beschrieben. Abbildung 4.5 zeigt eine beispielhafte Modellierung der Systemstruktur des technischen Systems einer Laboranlage. Das BDD beschreibt die strukturellen Zusammenhänge der Komponenten bzw. Subsysteme in dem System, das IBD auf der rechten Seite verfeinert alle relevanten Verbindungen, Daten- und Flussinformationen zwischen den vorhandenen Komponenten und definiert die nach außen sichtbaren Schnittstellen. Die Qualität der Modellbildung ist stets von dem zu entwickelnden System und den daran gestellten Anforderungen abhängig. Eine allgemeine Bewertung ist nur auf abstrakter Ebene und für ausgewählte Aspekte praktikabel [WBD10], [DWP07]. Die Erfüllung der Anforderungen aus Kapitel 2.4.2 schränkt die zur Modellierung des technischen Systems angewandte Methodik nicht weiter ein.

Die Beschreibung des Verhaltens von Komponenten und komponentenübergreifender Zusammenhänge erfolgt in Form von Zustands- und Aktivitätsdiagrammen. Katzke [Kat09] konnte zeigen, dass sich eine auf die Bedürfnisse des Maschinen- und Anlagenbaus reduzierte Beschreibungsmenge der UML, positiv auf die Informationsgewinnung auswirkt. Anwendungsfalldiagramme dienen primär zur Anforderungserhebung und als Diskussionsgrundlage und sind somit nicht zur Verhaltensbeschreibung von Komponenten notwendig. Sequenzdiagramme beschreiben eine konkrete Interaktion eines Szenarios und sind deshalb besonders für die Spezifikation von Testfällen [KTVH12] und nur begrenzt zur umfassenden Beschreibung von Systemverhaltensaspekten in frühen Phasen geeignet [HWs10].

Das Zustandsdiagramm bietet domänenspezifisch eine intuitive Beschreibungsform

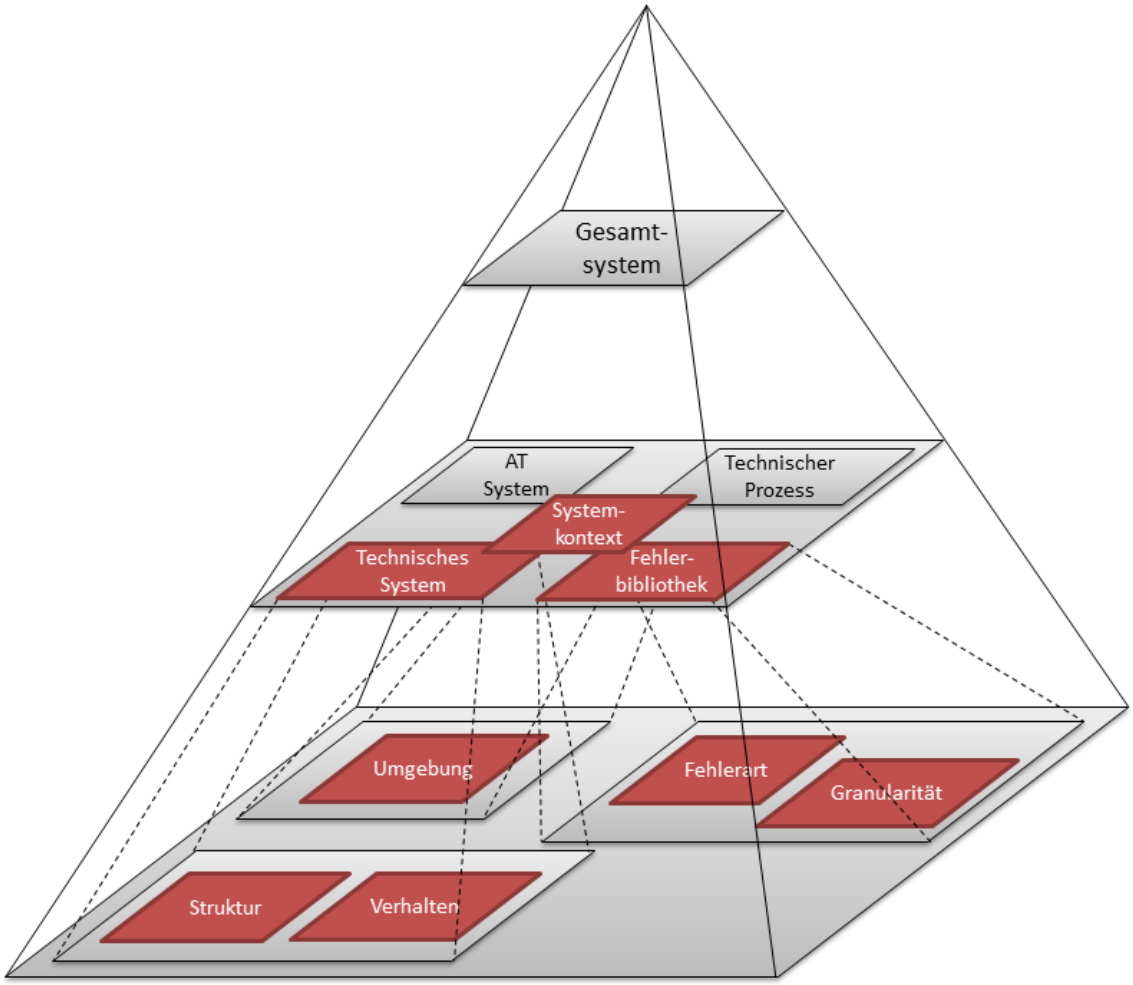


Abb. 4.4: Hierarchischer Aufbau und Bestandteile des gesamten Modellierungsansatzes

für das Verhalten von Komponenten, da insbesondere bei technischen Systemen Verhaltensspezifikationen in Form von endlichen Zustandsautomaten existieren. Dabei werden die Primärzustände des Systems auf Zustände im Modell abgebildet und deren Beziehungen zueinander in Form von Transitionsbeziehungen notiert. Durch die hohe Ausdrucksstärke des Zustandsdiagramms können auch komplexe Verhaltensmuster wie parallele Abarbeitung, Betriebsarten und Ausnahmesituationen (bspw. Not-Aus) direkt umgesetzt werden [WVH11], [VHBKF11]. Abbildung 4.6 zeigt exemplarisch das zustandsbasierte Verhalten eines Transportkrans.

4.2.1.2 Systemkontext

Der Systemkontext betrachtet das zu entwickelnde System als Black Box und setzt dieses zusammen mit den Schnittstellen und den Akteuren des System in Bezug zu seiner Umgebung. Durch die Integration des Systemkontextes in das Gesamtmodell können umgebungsrelevante Aspekte, die unter gewissen Umständen einen Einfluss auf die Zuverlässigkeit des Systems haben (bspw. Luftfeuchtigkeit der Umgebung),

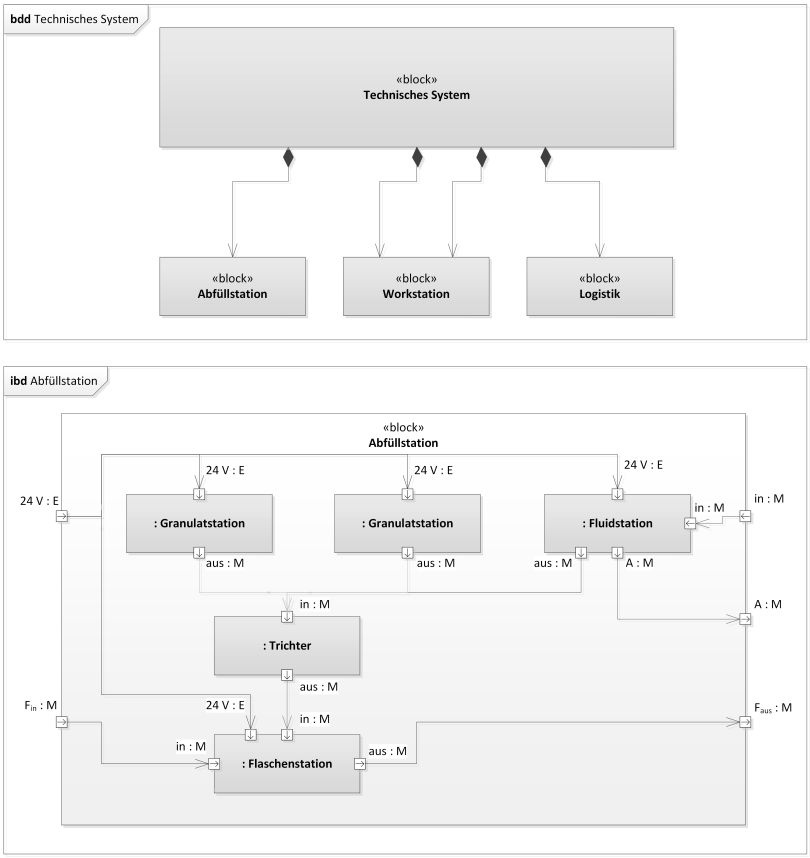


Abb. 4.5: Strukturmodellierung eines technischen Systems, oben: Systemstruktur (BDD), unten: Komponentenstruktur (IBD)

adressiert und zur davon abhängigen, integrierten Fehlermodellierung berücksichtigt werden [Wei08]. Abbildung 4.7 zeigt den Systemkontext einer Webmaschine. Die zum Betrieb der Webmaschine notwendige Strom- und Druckluftversorgung fließt dem System ebenso zu wie Kontrolleingaben zur Musterwahl. Diese Parameter können i.a. standortübergreifend stabil gehalten werden, jedoch trotzdem eine Auswirkung auf die Inbetriebnahme und den fortlaufenden Betrieb des Systems haben. Die Temperatur und die Luftfeuchtigkeit der Umgebung können je nach Standort des Systems stark variieren und können sich zudem im Laufe des Betriebs über die Zeit verändern und so unmittelbar Einfluss auf die Qualität des produzierten Gutes sowie auf die Stabilität und Zuverlässigkeit einzelner Komponenten und somit das Gesamtsystem nehmen. Das Systemkontextdiagramm integriert einflussstarke, systemrelevante Parameter durch die Einbeziehung der Umgebungsfaktoren in das Gesamtmodell.

4.2.2 Architektur und Integration des Fehlermodells

Die Zuverlässigkeit eines Systems kann durch die Präsenz von Fehlern negativ beeinflusst werden. Um die Auswirkungen im Fehlerfall zu minimieren bzw. deren Auftreten durch geeignete Gegenmaßnahmen zu unterdrücken, haben sich u.a. Maßnahmen der Fehlertoleranz etabliert. Dies kann u.a. durch die Integration von Fehlerbehandlungs-

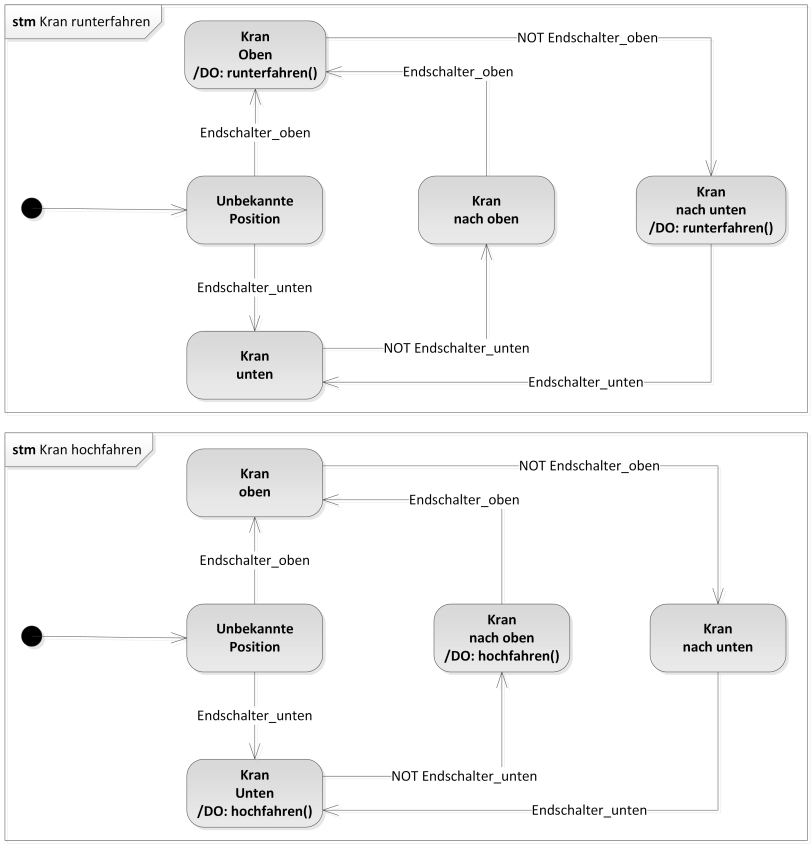


Abb. 4.6: Verhalten eines Transportkrans als Zustandsdiagramm, oben: Aktion runterfahren, unten: Aktion hochfahren

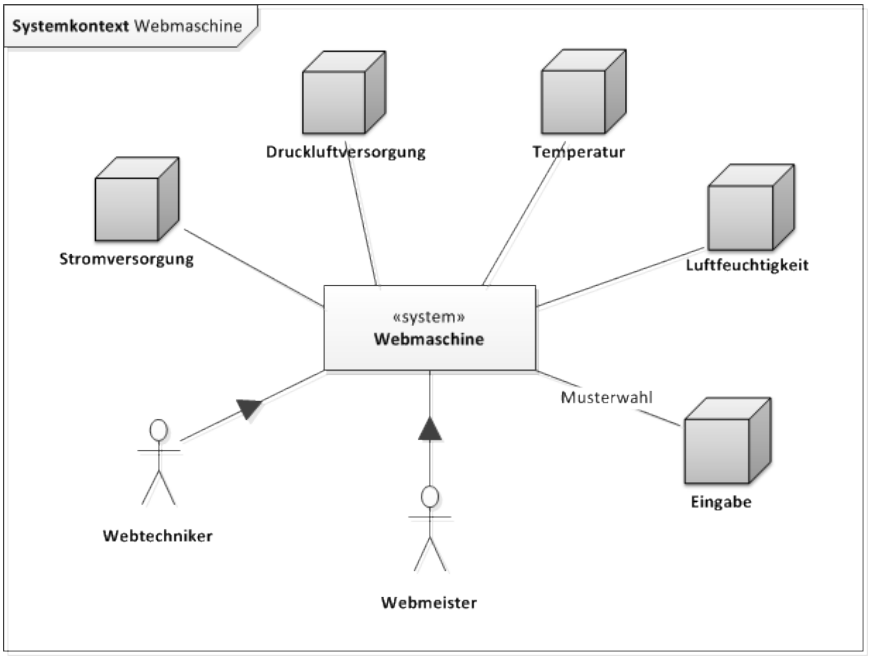


Abb. 4.7: Externe Einflussfaktoren auf das System Webmaschine in Form eines SysML Systemkontexts

mechanismen wie Fehlerabschaltung, Fehlerwiederherstellung und Fehlerkompensation erreicht werden [Lap95]. Wenngleich diese Mechanismen seit langem in der Steuerungssoftware eingesetzt werden, finden sie in der Testdurchführung nicht genügend Beachtung. Das lässt sich u.a. darauf zurückführen, dass die Fehler entweder nur sehr aufwändig zu provozieren [LW95] oder nicht explizit bekannt sind. Um dies zu adressieren, wird in diesem Unterkapitel ein Konzept zur Beschreibung und Integration von permanenten mechanischen Komponentenfehlern (**/SA2**), in das in 4.2.1 eingeführte Modell des technischen Systems vorgestellt. Je nach Durchdringungsgrad der Komponenten und den nichtfunktionalen Anforderungen an die zu entwickelnde Maschine bzw. Anlage liegen die Erkenntnisse über die Fehlerzusammenhänge in unterschiedlichen Detailstufen vor. Das eingeführte Fehlermodell muss demnach eine zweigliedrige Modellierungstiefe unterstützen (**/SA1**).

4.2.2.1 Gesamtüberblick der Fehlerbeschreibung

Die Erarbeitung einer Beschreibung und anschließenden Integration von Komponentenfehlern in ein Systemmodell bedarf einer präzisen Analyse der Fehlercharakteristika. Bei Fehlern handelt es sich, wie in Kapitel 3.2 eingeführt, stets um eine relative Größe, d.h. meist eine Abweichung im Sinne der Erwartung bzw. Forderung. Folglich kann ein Fehler als eine erweiterte (detailliertere) Beschreibung des Verhaltens betrachtet werden, das auf externen und i.d.R. physikalischen oder menschlichen Einflüssen beruht. Das Eintreten dieses speziellen Komponentenverhaltens kann letztendlich zu Systemanomalien führen und schwerwiegende Konsequenzen nach sich ziehen. Aus diesem Grund bedarf es der expliziten Berücksichtigung im Systemmodell, so dass die Systemreaktion auf diese Verhaltenssituationen während der Testphase explizit verifiziert werden kann.

Das Verhalten und die Zusammenhänge von Fehlern können demnach mit den in der SysML vorhandenen Struktur- und Verhaltensdiagrammen beschrieben und somit nahtlos in das SysML Gesamtmodell, siehe Kapitel 4.2.1, integriert werden. Die spezielle Ausprägung und somit Abweichung des Normalverhaltens einer Komponente kann als Attributierung bzw. attributierter Pfad betrachtet werden.

Die Verhaltensbeschreibung des Fehlermodells einer Komponente und das Steuerungsprogramm sind über das I/O Mapping bidirektional miteinander verbunden. Für die Generierung der Testeingabedaten bezüglich eines potentiellen Fehlers ist die tatsächliche Ausführungssequenz in der Verhaltensbeschreibung des Fehlermodells irrelevant. Die generierten Testeingabedaten stellen die Stimuli für die Steuerungssoftware dar, so dass die Komponente mit dem potentiellen Fehler entsprechend angesprochen wird. In welcher Form der potentielle Fehler im Rahmen der Testdurchführung provoziert wird und wie die Testumgebung, -ausführung und -bewertung zu behandeln sind, bedarf weiterer Forschung und ist nicht Gegenstand dieser Arbeit.

4.2.2.2 Zustandsdiskretisierte Modellierung von Komponentenfehlern

Ein gängiges Verfahren, kontinuierliches Systemverhalten zu modellieren, ist die Abstraktion und damit verbundene Diskretisierung der relevanten Zustände zur Beschreibung bzw. Approximation des Verhaltens, wie es bspw. bei Discrete Event Simulationen

zur Anwendung kommt [Ban09]. Diese abstrakte Form der Systemmodellierung lässt sich analog auf die Modellierung von Fehlern, respektive ihren wesentlichen Zuständen anwenden.

Zur Beschreibung des Fehlverhaltens auf Komponentenebene eignet sich das Zustandsdiagramm der SysML, wie in Abbildung 4.8 gezeigt. Der Normalablauf, bestehend aus den bereits modellierten Zuständen und Transitionen, stellt hierbei die Basis dar, die um die relevanten diskreten Zustände erweitert wird. Die Bedingungen, die die Transitionen beeinflussen, bestehen dabei im Wesentlichen aus wechselnden Sensorwerten, Benutzeraktionen oder Prozessvariablen. Das um das Fehlerverhalten erweiterte Normalverhalten wird durch die diskretisierten Zustände realisiert, die über Fehlervariablen angesprochen werden. Der Typ der Fehlervariablen entspricht dabei der jeweiligen Fehlerursache.

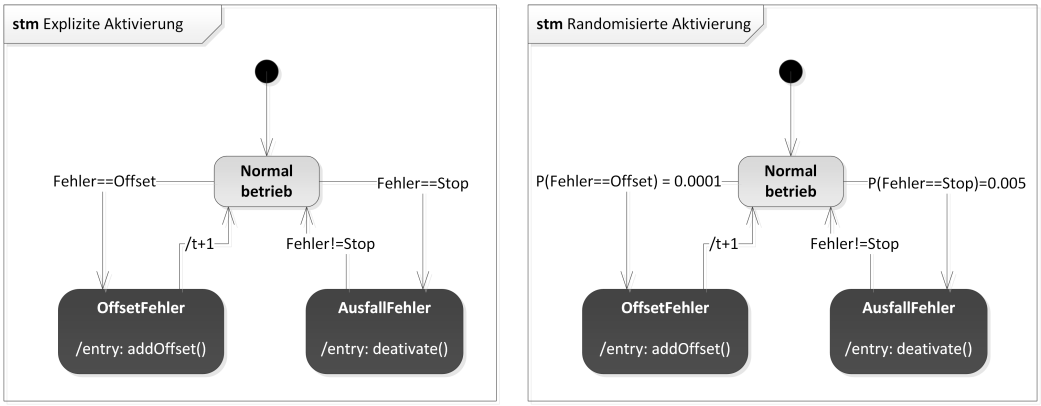


Abb. 4.8: Zustandsbasierte Fehlerbeschreibung mit expliziter (links) und randomisierter (rechts) Aktivierung

Als eine weitere mögliche Ausprägung der Fehlerzustandsaktivierung sind auch stochastische Transitionsbedingungen realisierbar, wie in Abbildung 4.8 rechts. Dies ist jedoch eine Frage der Verwendung des Fehlermodells. Im Gegensatz zur expliziten Aktivierung von Fehlerzuständen über boolesche Transitionsbedingungen können randomisierte Aktivierungsbedingungen auf pseudozufälliger Grundlage eingesetzt werden. Eine derartige Unterstützung zieht Auswirkungen der Ausführung nach sich und stellt somit zusätzliche Anforderungen an die Simulationsumgebung, in die das System- und Fehlermodell des technischen Systems eingebettet wird, vgl. Abbildung 4.2.

Bei der Identifikation und Beschreibung eines Fehlers ist es von elementarer Bedeutung, eine geeignete Modellierung der Fehlerzusammenhänge zu erzielen. Dabei sind besonders die Wahl des Zustandes, dessen Aktivierung und dessen Verlassenskriterium geeignet zu wählen. Ein zustandsbasiertes Fehlermodell wird nach den folgenden Regeln erstellt:

- **Fehlerzustand**: Ein Zustand wird genau dann zu einem Fehlerzustand, wenn sich während der Verweildauer in diesem Zustand eine wesentliche Eigenschaft der Komponenten verändert, bspw. Veränderung eines Reibungskoeffizienten.
- **Aktivierung**: Die zur Aktivierung des Fehlerzustandes notwendige Bedingung korrespondiert mit der Fehlerursache. Im Fall der expliziten Aktivierung ent-

spricht diese der booleschen Variablenexpansion. Bei einer randomisierten Aktivierung entspricht die Übergangsbedingung zur Aktivierung einer Wahrscheinlichkeit.

- **Verlassen**: Der Fehlerzustand wird stets in den Zustand zurück verlassen, aus dem der Fehlerzustand aktiviert wurde. Dabei können je nach Art des Fehlers Bedingungen an die Transition gekoppelt werden, wie bspw. eine Verzögerungszeit. Handelt es sich bei dem modellierten Fehler um einen in der Auswirkung über die Zeit zunehmenden Fehler, so kann die Transition derart formuliert werden, dass der Fehlerzustand mehrfach durchlaufen wird.

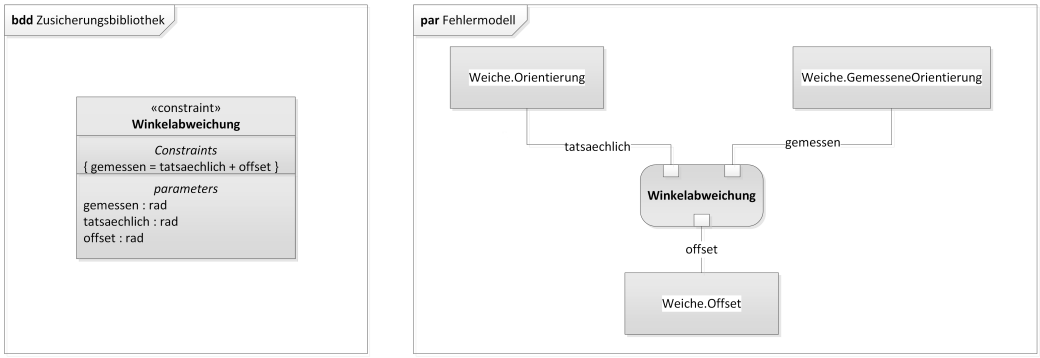


Abb. 4.9: Fehlerbeschreibung in Form eines Constraints (links) und deren Kombination mit den Parametern der Komponente als Zusicherung (rechts)

Die Anforderung **/SA2** an die System- und Fehlermodellierung bezieht sich auf mechanische, permanente Fehler. Ein im System permanent existierender Fehler zeichnet sich dadurch aus, dass dessen Wirkung gleich bleibend, aber auch über die Zeit variierend sein kann. Die Modellierung gleich bleibender Fehler muss derart gestaltet werden, dass der Fehlerzustand entweder exakt einmal durchlaufen wird oder bei mehrfacher Aktivierung keine Verhaltensänderung nach sich zieht. Bei variierenden Fehlern kann der entsprechende Zustand mehrfach durchlaufen und dessen Wirkung dadurch intensiviert werden. Dies ist üblicherweise bei systematischen Fehlern wie Abnutzung, Korrosion etc. der Fall.

4.2.2.3 Fehlermodellierung auf Basis physikalischer Zusammenhänge

Das Konzept der Modellierung von Komponentenfehlern sieht eine bedarfsgerechte Unterstützung der Detailtiefe bzw. Granularität vor (**/SA1**). Der in Kapitel 4.2.2.2 eingeführte Modellierungsansatz der Zustandsdiskretisierung eignet sich für die unmittelbare Beschreibung von grundlegenden Fehlerzusammenhängen. Dabei kann die Qualität der Beziehungen insbesondere über den Umfang der abgebildeten Attribute skaliert werden. Bei komplexen Fehlern kann dies einerseits zu einer unverhältnismäßig hohen Anzahl an Hilfszuständen und Transitionsbedingungen einerseits, aber auch einer unpräzisen Fehlerrepräsentation andererseits führen.

Komplexere Zusammenhänge lassen sich meist in Form von physikalischen (mathematischen) Beziehungen ausdrücken und bedürfen einer adäquaten Beschreibungsform. Insbesondere basieren Eigenschaften mechanischer Komponenten auf physikalischen

Gesetzmäßigkeiten, deren Ursache und Wirkung sich unmittelbar mit Hilfe mathematischer Formeln beschreiben lassen. Da es sich bei Fehlern um spezielle Eigenschaften von Komponenten handelt, können die physikalischen Eigenschaften als beeinflussende, einschränkende Beziehungen zwischen den Attributen einer Komponente interpretiert werden.

In der SysML existiert ein Zusicherungsmechanismus zur Definition von parametrischen Beziehungen zwischen Eigenschaften der vorhandenen Systembausteine. Die Eigenschaften reichen dabei von grundlegenden Zusammenhängen, wie bspw. dem Newtonschen Gesetzen bis zu Einschränkungen in den Verhaltensaspekten, wie bspw. der Rotationsauflösung eines Motors. Dieser Mechanismus bietet die Möglichkeit der Beschreibung und somit auch Integration von Fehlermodellen. Durch den Einsatz einer speziellen Blockdefinition, dem sog. Constraint, werden Zusicherungen an Systemstrukturen mit den dazugehörigen Parametern in einem Zusicherungs- oder Constraint-Block definiert, siehe Abbildung 4.9 links.

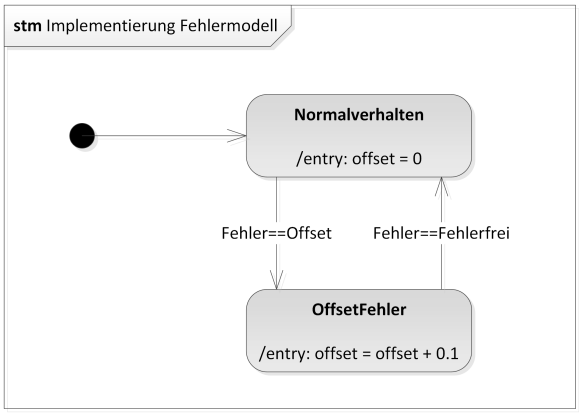


Abb. 4.10: Verhaltensimplementierung des parametrischen Fehlermodells

Constraints dieser Art beschreiben physikalische Gesetzmäßigkeiten, die in Form einer Modellbibliothek einmal erstellt und zur Wiederverwendung abgelegt werden können (**/SA3**). Im SysML Zusicherungsdiagramm (auch Parameterdiagramm genannt) werden die Constraints aufgegriffen und deren Parameter über sog. Konnektoren miteinander gekoppelt, siehe Abbildung 4.9 rechts. Darüber erfolgt eine präzise mathematische Modellierung permanenter, mechanischer Fehler. Über die intrinsischen Zusammenhänge wird letztendlich das korrekte Fehlerverhalten auf Basis der modellierten physikalischen Grundsätze induziert. Zur Anwendung dieses Mechanismus' für Fehlermodelle müssen dem jeweiligen Systembaustein (Komponente) im Blockdefinitionsdiagramm die zur Fehlerbeschreibung relevanten Parametersätze hinzugeführt werden. Die Implementierung des dazugehörigen Verhaltens erfolgt weiterhin im Zustandsdiagramm und kann somit als Erweiterung der reinen Zustandsdiskretisierung eingesetzt werden.

Abbildung 4.10 zeigt die zur Abbildung 4.9 korrespondierende Verhaltensimplementierung für einen Offsetfehler. Die Umsetzung im Form des Zustandsdiagramms unterscheidet sich im Vergleich zur Zustandsdiskretisierung dahingehend, dass die Abhängigkeiten bzw. physikalischen Zusammenhänge der Parameter getrennt von der reinen Verhaltensmodellierung berücksichtigt werden. Dies führt dazu, dass grund-

sätzliche Zusammenhänge in Form einer Modellbibliothek wiederverwendet werden können und somit eine aufwandsarme Modellierung ermöglicht. Die Trennung von Komponentenverhaltensimplementierung und den intrinsischen Parametereigenschaften, unterstützt hier zusätzlich eine Skalierung der Fehlergranularität ohne jegliche Modifikation der Verhaltensimplementierung. Letztendlich wird die Fehlerkomplexität von der Beschreibung im Zustandsdiagramm entkoppelt, was zu einem geringeren Modellierungsaufwand und aussagekräftigeren Verhaltensbeschreibungen führt.

4.2.2.4 Integration des Fehlermodells in das Gesamtmodell

Durch den Einsatz des Drei-Sichten-Konzepts basiert das in diesem Kapitel vorgestellte Konzept der fehlerorientierten Systemmodellierung auf einer einheitlichen Modellierungssprache und unterstützt eine integrierte Modellierung. Die Erweiterung um Fehleraspekte erfolgt mit den vorhandenen SysML Diagrammen. Physikalische Zusammenhänge werden unter Verwendung des Zusicherungsmechanismus nahtlos in das Gesamtmodell integriert und bedürfen keiner weiteren Modifikation zur durchgängigen Modellbildung.

Bei der Integration des Fehlermodells in das Gesamtmodell ist eine geeignete Verschmelzung der Primärfunktionalität mit dem attributierten Verhalten im Fehlerfall ausschlaggebend. Dabei handelt es sich bei dem präsentierten Konzept um die Verhaltensbeschreibung in Form des Zustandsdiagramms, sowohl bei der Zustandsdiskretisierung aus Kapitel 4.2.2.2 als auch der Fehlermodellierung auf Basis physikalischer Zusammenhänge aus Kapitel 4.2.2.3. Zur Beschreibung des Komponentenverhaltens wird das Zustandsdiagramm eingesetzt. Das attributierte Fehlerverhalten einer Komponente kann als natürlicher Prozess betrachtet werden, der basierend auf den äußeren Umwelteinflüssen, parallel und zu jeder Zeit, auf das Gesamtsystem und somit jede einzelne Komponente einwirkt. Orthogonale Zustände bieten die Ausdrucksmöglichkeit parallel laufender Prozesse im Zustandsdiagramm. Dadurch kann jede Primärfunktionalität des Zustandsdiagramms mit der orthogonalen Erweiterung um die Fehlerfunktionalität ergänzt werden. Aufgrund der integrierten, durchgängigen Systemmodellierung ist eine Konsistenz der Modelle gewährleistet und die Erweiterung aufwandsarm realisierbar.

Die Integration des Verhaltensmodells als Erweiterung in eine orthogonale Region des Zustands ist in Abbildung 4.11 gezeigt. Der Ablauf der Primärfunktionalität wird dabei aus der existierenden Umsetzung übernommen und um das Fehlermodell parallel erweitert. Durch die Kopplung über den Zusicherungsmechanismus werden die gezeigten Constraints aus Abbildung 4.9 der Umsetzung zugrunde gelegt.

Das Konzept der Fehlermodellierung und Integration in das Systemgesamtmodell wird der im nächsten Unterkapitel folgenden Testdatengenerierung zugrunde gelegt, so dass ein zielgerichtetes und testdatenreduzierendes Verfahren realisiert werden kann.

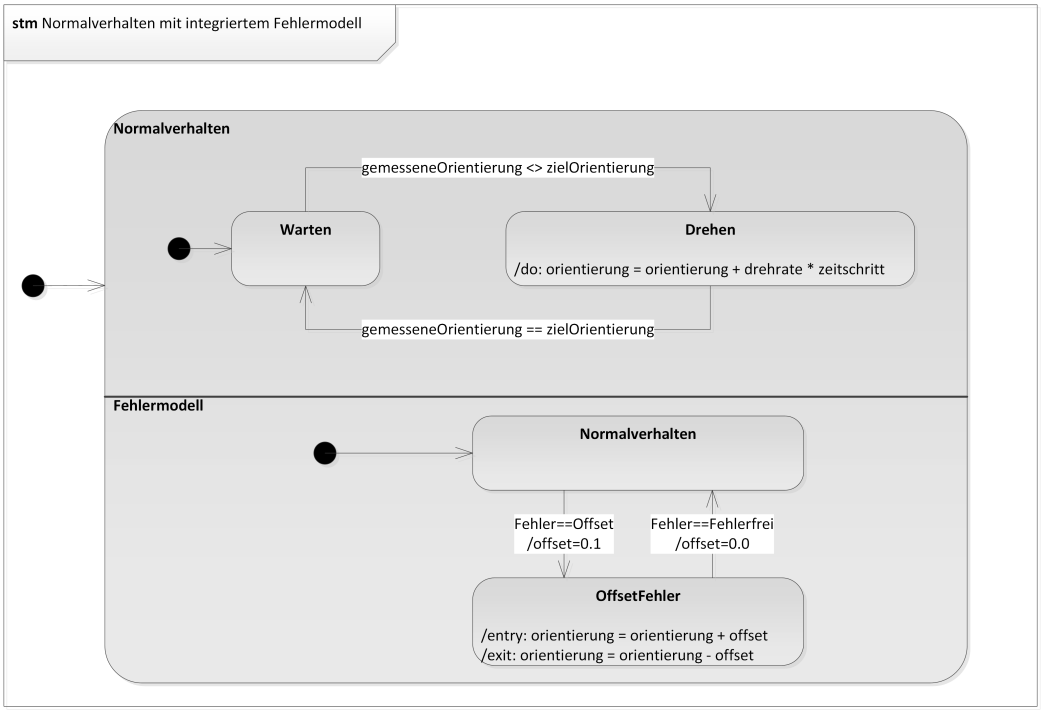


Abb. 4.11: Integration des Fehlerverhaltensmodells in das Gesamtverhalten

4.3 Allgemeines Konzept der Testdatengenerierung

Bei der Durchführung von Tests bestimmen die gewählten Eingabedaten die Aussagekraft, Testabdeckung und Testtiefe des Testprozesses. Die Testeingabedaten sind wesentlicher Bestandteil der Test Case Specification [IEE08]. Dabei muss zur praktischen Realisierbarkeit einer geforderten Absicherung eine Optimierung zwischen hoher Testtiefe und zeitlicher Umsetzbarkeit stattfinden. Die Wahl der richtigen und wichtigen Parameter stellt demnach eine ebenso zentrale Herausforderungen dar wie der zeitliche Aufwand, diese abzuleiten.

Das Testziel ist dabei, ein zu entwickelndes System gegen potentielle Fehler zu prüfen und dessen Gesamtverhalten zu bewerten. Das in diesem Kapitel präsentierte Konzept der Testdatengenerierung ist ein automatisches Verfahren zur Ableitung von Testdaten basierend auf einer Strukturanalyse eines gegebenen IEC 61131-3 Quelltextes. Durch die Integration von mechanischen Fehlern in Form eines Fehlermodells im Systemgesamtmodell, siehe Kapitel 4.2, kann eine automatische zielgerichtete Ableitung der Testdaten erfolgen. Die Dynamik der Steuerungssoftwareentwicklung führt meist zu Fragmenten des Quelltextes, die

- vermutlich niemals ausgeführt werden, aber trotzdem zu einer bestimmten Zeit aktiviert werden und somit ein unbekanntes Systemverhalten nach sich ziehen
- entworfen wurden, um die Fehlertoleranz zu erhöhen, jedoch niemals ausgeführt werden, weil der dazugehörige Systemzustand oder der jeweilige Abschnitt des Quelltextes niemals erreicht wird

All diese Situationen erfordern ein quelltextbasiertes Testdatengenerierungsverfahren und einen automatisierten Prozess der Generierung, um Abdeckungskriterien und somit die geforderte Software- und Systemqualität nachzuweisen.

4.3.1 Problembeschreibung bei zyklischer Software

Die zyklische Ausführungslogik der IEC 61131-3 basierten SPS Programme erfordert je nach POE einer Applikation ein neuartiges Generierungsverfahren. Insbesondere bei Funktionsbausteinen, siehe Kapitel 4.2.1 schlagen bisherige Generierungsansätze fehl.

4.3.2 Beschreibung des Lösungsansatzes

Die Steuerungssoftware interagiert mit dem System- und Fehlermodell über ein bidirektionales I/O Mapping, siehe Abbildung 4.2. Jeder mögliche und beabsichtigte Komponentenfehler kann Konsequenzen nach sich ziehen, nachdem die Funktionalität dieser Komponente durch einen Abschnitt der Steuerungssoftware angesprochen wurde. Dieser bestimmte, mit der fehlerhaften Komponente kommunizierende Abschnitt des Quelltextes kann mehrere Eingabeparameter mit unterschiedlichen Wertebereichen annehmen. Der relevante Abschnitt des Quelltextes wird jedoch nur erreicht und ausgeführt, wenn die Eingabeparameter entsprechend kombiniert werden. Aufgrund der Quelltextkomplexität ist eine manuelle Ableitung der korrekten Eingabewerte eine herausfordernde und zeitintensive Aufgabe.

Eine Simulationsumgebung mit integrierten Fehlermodell steht in direkter Interaktion mit dem Steuerungscode. Dadurch repräsentieren die Ausgabewerte der Simulationsumgebung die Eingabewerte des Steuerungscodes [KS16]. Dieses Setup kann für sog. Hardware-in-the-Loop (HiL), aber auch Software-in-the-Loop (SiL) Tests verwendet werden. Ziel ist es nun, die Simulationsparameter, die als Eingabedaten auf den Steuerungscode wirken, für Tests von Fehlerszenarien in einem Offline-Verfahren zu ermitteln. In Abbildung 4.12 ist ein Ausschnitt einer konfigurierbaren TrySim[1] Simulation für ein reales System dargestellt.

Abbildung 4.13 zeigt den allgemeinen Aufbau und Ablauf des Testgenerierungsverfahrens. Die Steuerungssoftware wird in ein Zwischenformat, den Kontrollflussgraphen transformiert, aus dem alle notwendigen Informationen zur Testdatengenerierung extrahiert werden. Die dadurch ermittelten Testdaten dienen der Stimulation der Steuerungssoftware. Bei einer Testdurchführung muss anhand dieser Daten das System (die Simulation) in den Zustand versetzt werden, so dass der Fehler in das Gesamtsystem injiziert werden kann. Die Ausführung und Fehlerinjektion ist nicht mehr Gegenstand dieser Arbeit.

Unter Verwendung der sog. symbolischen Ausführung wird der Quelltext formal analysiert und durch mathematische Auswertung der Ausdrücke die konkreten Eingabeparameter automatisch abgeleitet werden. Dazu werden zwei Ansätze verfolgt.

1 TrySim ist ein Simulationswerkzeug, das Kopplungen für zahlreiche Steuerungshersteller anbietet. Es eignet sich besonders für die Simulation von Abläufen und die Betrachtung von Fehlern in technischen Komponenten.

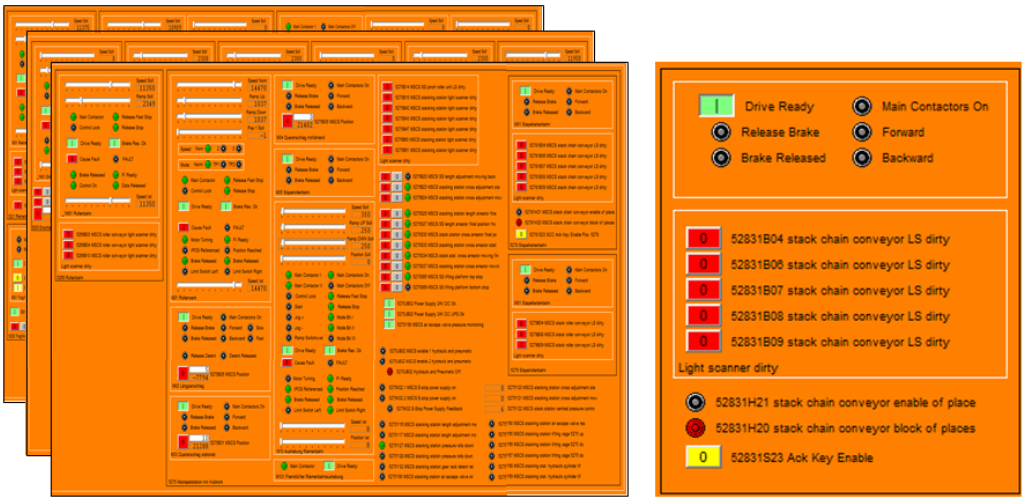


Abb. 4.12: Beispielhafter Simulationsumfang für die Prüfung von Fehlern

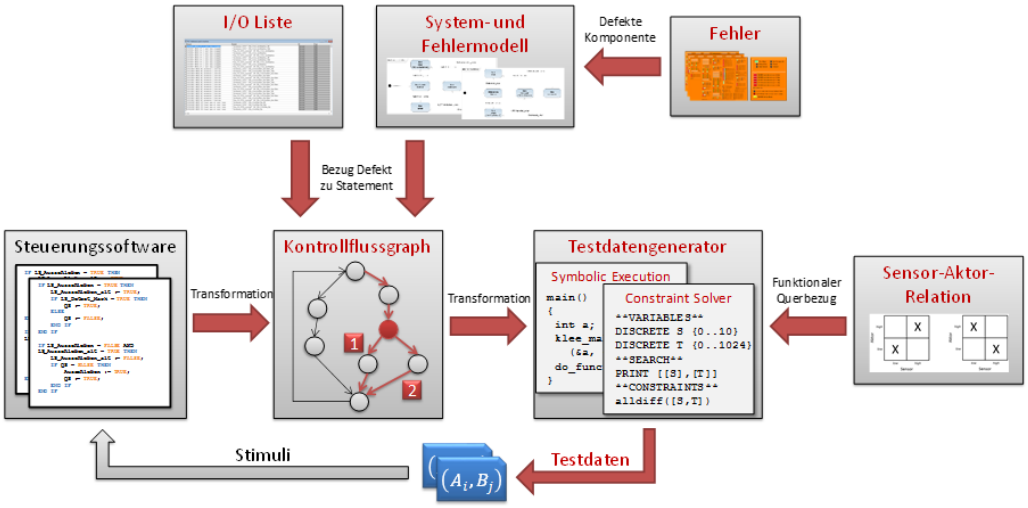


Abb. 4.13: Beteiligte Komponenten und Ablauf der Testdatengenerierung

Im Abschnitt 4.4 wird der Ansatz erläutert, worin die Informationen der Steuerungssoftware unter Verwendung eines Constraint Solvers ausgewertet und durch explizite Betrachtung der Datenabhängigkeit zwischen disjunkten Pfaden der Testeingabedatensatz ermittelt wird. Anweisungen und Bedingungen werden dabei als Pfadrestriktionen des Kontrollflusses interpretiert. Die gefundene Lösung, die alle Restriktionen erfüllt, beinhaltet alle gültigen Eingabewerte. In zweiten Ansatz unter Abschnitt 4.5 wird der Steuerungsquelltext so transformiert, so dass dieser mit einem Werkzeug zur symbolischen Ausführung von C-Programmen ausgewertet werden kann. Da diese Werkzeuge keine zyklische Ausführungslogik unterstützen, wie in Abschnitt 3.4.2.3 erläutert, wird eine weitere Abbildungsregel eingeführt.

SPS Applikationen können unabhängig der POEs in den fünf verfügbaren IEC 61131-3 Programmiersprachen realisiert werden. Eine wesentliche Eigenschaft von SPS Entwicklungsumgebungen ist die Abbildung aller SPS Programmiersprachen auf ein

gemeinsames Zwischenformat. Der Übersetzungs- und Optimierungsvorgang in lad- und ausführbaren Maschinencode, auf einem Automatisierungsgerät erfolgt auf diesem gemeinsamen Zwischenquelltext. Dies ermöglicht u.a. die Erweiterung um zusätzliche Programmiersprachen, wie UML [WVH11], [WRK10], die durch eine Abbildungsvorschrift auf ST erreicht wird. Das in diesem Kapitel präsentierte Konzept wird auf ST Sprachkonstrukte optimiert, ist jedoch grundsätzlich auf jegliche Programmiersprachen anwendbar.

4.4 Explizite Betrachtung der Datenabhängigkeit

Die Erreichbarkeit von Anweisungen (Pfad) in zyklisch ausgeführten Bausteinen mit Gedächtnis, d.h. internem Zustand, hängt nicht von einem einzigen Eingabedatensatz ab. Ein Pfad ist eine eindeutige Sequenz von Zweigen einer Funktion, von deren Beginn bis zu deren Ende [Bei90]. Bei unendlich häufig ausgeführten Bausteinen, gibt es somit theoretisch unendlich viele Pfade. Für die Bestimmung des Pfades zur Erreichbarkeit einer Anweisung bedarf es demnach einer mehrfachen Ausführung bzw. im Rahmen des Tests, einer mehrfachen Stimulation des Testobjekts. In Abbildung 4.14 ist die Berücksichtigung der Datenabhängigkeit in verschiedenen Zweigen für die Generierung von Constraint Solving Problemen dargestellt. Für die Erreichbarkeit des Knotens 7 muss eine Entscheidung in Knoten 2 und 5 wahr werden. Die Bedingung in Knoten 2 hängt jedoch von Knoten 4 ab, so dass Knoten 7 nur erreicht werden kann, sofern Knoten 4 vorher durchlaufen wird. Diese Datenabhängigkeit wird verwendet, um die (grundsätzlich) unendliche Schleife auszurollen. Aus den resultierenden sequentiellen Schritten werden jeweils Constraint Solving Probleme generiert.

4.4.1 Grundlagen Constraint Solving

In der Informatik/Mathematik existieren zahlreiche Ansätze zur Lösung von Suchproblemen, d.h. dem Aufdecken von gültigen Lösungen eines i.a. mathematischen Problems. Durch den Einsatz von Suchalgorithmen können Lösungen problemunabhängig gemäß der algorithmischen Komplexität gelöst werden. Aufgrund dieser Einfachheit erfolgt deren Anwendung bei monokriteriellen Strukturabfragen.

Das Constraint Satisfaction Problem (CSP) beschreibt ein komplexes Suchproblem, bei dem eine Menge struktureller Einschränkungen den Lösungsraum begrenzt. Zur Erfüllung der Zielbedingung müssen alle Teilbedingungen erfüllt werden. Die formale Definition des CSPs, für die Testdatengenerierung ist in Definition 9 gegeben.

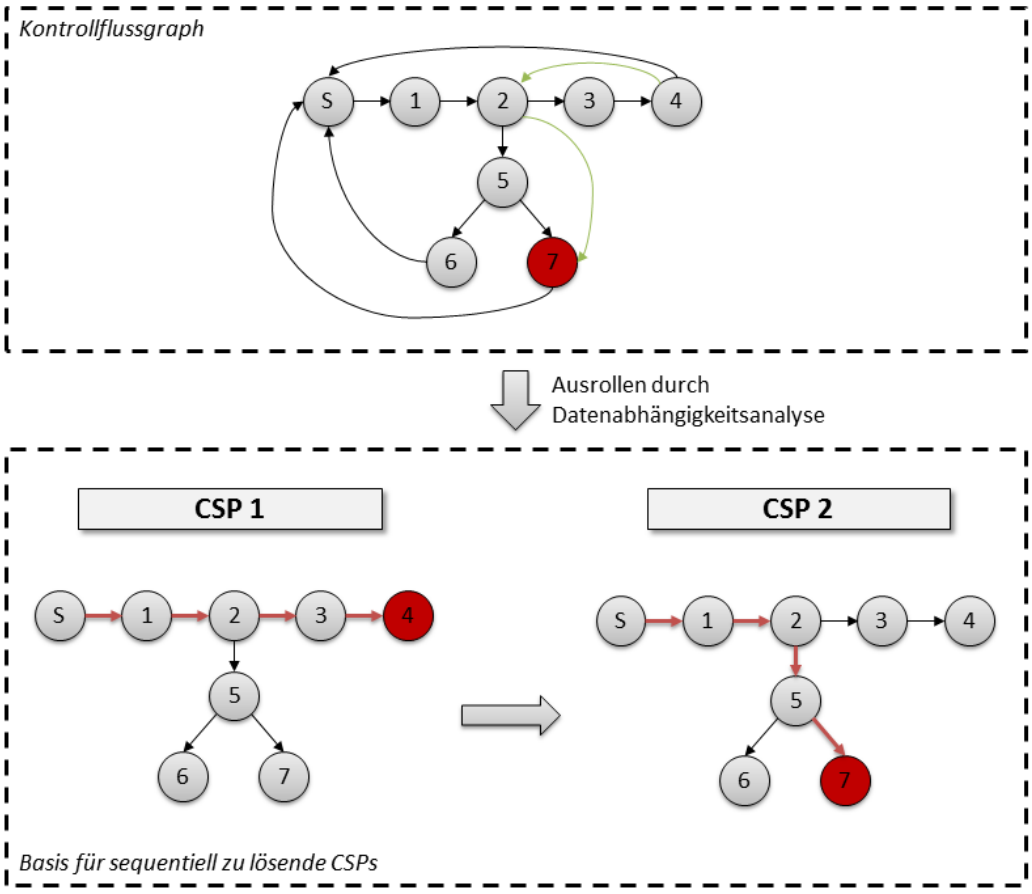


Abb. 4.14: Ableiten der CSPs auf Basis des ausgehenden Kontrollflussgraphen unter Berücksichtigung der Datenabhängigkeit. Rot symbolisiert den zu erreichenden Knoten inklusive Pfad. Grün zeigt die Datenabhängigkeit zwischen Knoten.

Definition 9 (*Constraint Satisfaction Problem (CSP)*)

Ein Constraint Satisfaction Problem CSP (X,D,C) *besteht aus einer Menge von Variablen* $X = \{x_1,x_2,\ldots,x_n\}$ *mit den dazugehörigen endlichen Wertebereichen* $D_1,D_2,\ldots,D_n$. *Die Menge von Constraints* C *begrenzt die Werte von* X. *Die Constraint Satisfaction Aufgabe ist es, gültige Werte für* X *innerhalb* D *zu finden, so dass alle Constraints in* C *erfüllt sind. Ein Constraint* $c \in C$ *ist eine Relation* R *über* $k \leq n$ *Variablen. Das Prädikat* $\delta(x_1,x_2,\ldots,x_k)$, *definiert über das kartesische Produkt der korrespondierenden Wertebereiche, muss evaluiert werden. Wenn alle Constraints erfüllt sind, so dass* $(\delta(x_1),\ldots,\delta(x_k)) \in R$ *gilt, existiert eine gültige Lösung des CSPs.*

Ein typisches Applikationsbeispiel des CSP ist das N-Damenproblem, bei dem N identische Damen so auf einem $N \times N$ Schachbrett platziert werden müssen, so dass sie sich gegenseitig nicht schlagen können. Dabei muss gelten, dass keine zwei Damen sich in derselben Spalte, noch in derselben Zeile oder entlang einer Diagonalen befinden. Bei der mathematischen Beschreibung eines Sachverhaltes kann die Wahl der Codierung, d.h. der Abbildung der Realität auf die Mathematik, einen entscheidenden Einfluss

auf die Problemkomplexität und somit auch auf die Lösung haben. Nach [RN10] genügen zur Codierung des N-Damenproblems n Variablen $X = \{x_1,x_2,\ldots,x_n\}$ mit dem dazugehörigen Wertebereich $\{D_1,D_2,\ldots,D_n\}$. Dabei repräsentiert die Variable x_i die Spalte i und der Wert von $x_i \in D_i$ die jeweilige Zeile, auf der sich die Dame befindet. Diese Codierung beinhaltet implizit der Forderung, dass sich keine zwei Damen in einer Spalte befinden, indem explizit maximal eine Dame pro Spalte codiert wird. Die beiden weiteren Einschränkungen, dass sich keine zwei Damen entlang einer Zeile und entlang einer Diagonalen befinden dürfen, ist in den Constraintungleichungen 4.1 und 4.2 angegeben.

$$c_1 : \forall i,j; i \neq j : x_i \neq x_j \tag{4.1}$$

$$c_2 : |x_i - x_j| \neq |i - j| \tag{4.2}$$

Für das $n = 4$-Damenproblem ergibt sich demnach als eine gültige Lösung die folgende Belegung: $\delta(x_1 = 2,x_2 = 4,x_3 = 1,x_4 = 3)$.

Es lassen sich verschiedene Typen von CSPs unterscheiden, die sich in ihrer Komplexität und damit ihrer praktischen Berechenbarkeit und den damit einhergehenden Anforderungen an die Algorithmik verschieden sind. Die Suche nach Wertebelegungen, die alle Constraints erfüllt, ist NP-vollständig. Dabei können die Wertebereiche der Variablen als diskrete, kontinuierliche, endliche und unendliche Ausprägungen sowie lineare als auch nichtlineare Constraints auftreten. Für nichtlineare Constraints existieren keine Algorithmen zur Lösung. Variablen mit einem kontinuierlichen Wertebereich sind hauptsächlich in Anwendungen des Operations Research vorzufinden, bei der Probleme aus der Kategorie der linearen Programmierung in polynomialer Zeit lösbar sind [RN10]. CSPs mit diskreten Variablen und endlichen Wertebereichen repräsentieren die einfachsten Arten [RN10]. Das eingangs verwendete n-Damenproblem gehört dabei zu dieser Kategorie.

CSPs können besonders effizient für Probleme mit endlichen Wertebereichen angewendet werden. SPS Applikationen bestehen im Wesentlichen aus Variablen mit diskreten und endlichen Werten. Der Kontrollfluss wird durch Bedingungen auf den Variablen gesteuert, so dass je nach vorhandenen Eingabedaten der SPS Applikationen der dazu korrespondierende Pfad gewählt und ausgeführt wird. Demnach kann das Problem der Generierung von Testdaten als ein CSP formuliert werden.

Zur Lösung eines Constraint Satisfaction Problems existieren sog. Constraint Solver, die auf diversen algorithmischen Lösungsverfahren beruhen [ZY01]. Das in diesem Konzept vorgestellte Verfahren zur Generierung der Testeingabedaten zyklischer Softwarebausteine formulierte CSP wird direkt auf ein deklaratives Format eines Constraint Solvers abgebildet. Dazu wird der freie und Open Source General-Purpose Constraint Solver Minion [Min11] eingesetzt. Minion dient als Eingabe ein im Minionformat formuliertes CSP, welches gemäß der Formatspezifikation in Sektionen zur Beschreibung der Variablen und Wertebereiche, den Suchkriterien und den problemspezifischen Restriktionen (Constraints) eingeteilt ist. Eine detaillierte Beschreibung des Formats

erfolgt bei der Beschreibung der Generierungsregeln in Kapitel 4.4.2.2.

4.4.2 Generierung der Testdaten

In diesem Unterkapitel werden die Grundlagen der Testdatengenerierung von der formalen Basis bis zur Transformation des Ausgangsquelltextes zum Constraint Satisfaction Problem eingeführt. Der hier eingeführte Formalismus dient dem um die Datenabhängigkeit erweiterten Generierungsverfahren in Kapitel 4.4.3 als Grundlage.

4.4.2.1 Formale Basis

Eine Funktion oder ein Funktionsbaustein als Programmorganisationseinheit (POE) kann als ausführbarer Quelltextbestandteil F mit Variablen X, Anweisungen S und Bedingungen C betrachtet werden, die den Steuerungsfluss einer Applikationen bestimmen. In Analogie zu höheren Programmiersprachen wie `C++` besitzt jede Funktion eine formale Parameterliste mit den jeweiligen Eingabeparametern P_f und den dazu korrespondierenden Datentypen und jeweiligen Wertebereichen. Die Ausgabeparameter müssen für die Testdatengenerierung nicht berücksichtigt werden, da die gewählten Zweige eines Kontrollflusses nur von den konkreten Eingabewerten abhängig sind. Jedoch hängt der Kontrollfluss einer SPS Steuerungsapplikation nicht nur von den Eingabewerten der formalen Parameterliste ab, sondern ebenfalls von globalen Parametern und den I/O Eingabewerte P_{io}. Die Eingabeparameter in P_{io} können dabei als direkt gebundene Ausgabeparameter des Systemmodells betrachtet werden, siehe Abbildung 4.2.

Durch Techniken der statischen Codeanalyse erfolgt eine einmalige Bewertung des Quelltextes bzw. Kontrollflusses. Aufgrund der zyklischen Ausführungslogik von SPS Applikationen wird der gegebene Code gewöhnlich mehrfach ausgeführt. Da dieser Sachverhalt bislang nicht entsprechend berücksichtigt wird, kann es bspw. durch sich widersprechende Bedingungen, die auf prozess- bzw. systembezogene Variablen basieren, zum gegenseitigen Ausschluss kommen. Deshalb wird zur Lösung dieses möglichen Konflikts in diesem Konzept eine zeitdiskrete Strukturanalyse zur Testdatengenerierung eingeführt.

Definition 10 (*Ableiten von Testdaten*)

Die Ableitung von Testdaten zu einem gegebenen Quelltextfragment (Funktionsbaustein, Funktion) $F(P_f,P_{io},X,S,C)$, so dass eine bestimmte Anweisung $s \in S$ erreicht und ausgeführt wird, entspricht der gültigen Untermenge der Parameterbelegung von $P_f \cup P_{io}$, so dass alle Bedingungen $C' \subseteq C$ vom Start des Codes bis zum Erreichen von s wahr sind.

4.4.2.2 Formulierung des CSPs

Die Diversität der fünf IEC 61131-3 Programmiersprachen [IEC03a] wird durch die Transformation auf den Kontrollflussgraphen, siehe Abbildung 4.13, auf ein einheitliches Datenformat gebracht. Für die weitere Bearbeitung und Problemlösung dient der Abhängigkeitsgraph (Kontrollfluss) des Quelltextes. Dieses Zwischenformat dient

zusätzlich als Vorstufe für zahlreiche Compileroptimierungen, wie Constant Propagation während des Übersetzungsvorgangs [ASU86]. Dabei wird das etablierte Static Single Assignment (SSA) Format verwendet, bei dem Variablen über ihre gesamte Lebenszeit nur einmal Werte zugewiesen werden. Ein Kontrollflussgraph wird dabei wie folgt definiert:

Definition 11 (*Abhängigkeitsgraph (Kontrollfluss)*)

Ein Kontrollflussgraph $CFG(V,E,v_s,v_e)$ besteht aus einer Menge von Knoten V und gerichteten Kanten $E : V \times V$. Der Kontrollfluss verweist stets ausgehend vom Startknoten $v_s \in V$ entlang mindestens eines Pfades bis zum Endknoten $v_e \in V$. Des Weiteren gilt, dass der Startknoten keinen Vorgänger $\forall v_i \in V : (v_i, v_s) \notin E$ und der Endknoten keinen Nachfolger $\forall v_i \in V : (v_s, v_i) \notin E$ haben darf.

Die Transformation des SPS Steuerungsquelltextes in den Kontrollflussgraphen erfolgt gemäß den folgenden Vorschriften: Alle Anweisungen S und Bedingungen C des Quelltextes werden direkt als disjunkte Vereinigung auf Knoten $V = S \uplus C$ des Graphen G abgebildet. Dabei wird die erste Anweisung auf $v_i \in V$ und die letzte Anweisung von F auf $v_e \in V$ zugewiesen. Die geordnete Menge von Kanten E enthält alle Knotentupel $(v_i,v_j), i \neq j$, für die ein direkter Pfad der Länge 1 existiert, d.h. v_j ist der direkte Nachfolger von v_i. Dies gilt sowohl für alle Anweisungen als auch Bedingungen.

Die vollständige Struktur eines Quelltextes wird durch Kontrollstrukturen bestimmt. Davon abhängig muss auch der dazugehörige Kontrollflussgraph generiert werden. Tabelle 4.2 zeigt die Transformationen von Kontrollstrukturen auf dem Kontrollflussgraphen basierend auf den Syntaxgraphen der Programmiersprache ST für *Anweisungen*, *for*-Schleifen, *while*-Schleifen, *case*-Anweisungen und einer *if*-Anweisung. Auf Basis dieser elementaren Kontrollstrukturen lassen sich alle hierarchisch zusammengesetzten Kombinationen zusammenstellen und transformieren.

Der resultierende Graph repräsentiert alle Kontrollflüsse des ursprünglichen Quelltextes. Ein Kontrollfluss repräsentiert alle möglichen Pfade ohne Bezug zu den Daten. Die konkreten Werte von Parametern oder Variablen im Allgemeinen koordinieren den Fluss eines Programms. Folglich bedarf es zur Bestimmung der Testdaten und somit der Erreichbarkeit einer gegebenen Anweisung einer Relation zwischen den Eingabeparameterwerten und den daraus resultierenden Pfaden. Zur Berechnung der Parameterdaten werden die Informationen aus dem Kontrollflussgraphen in ein Constraint Satisfaction Problem transformiert, welches mit Hilfe eines Constraint Solvers gelöst werden kann.

Wie in Abbildung 4.15 gezeigt, müssen Eingabedaten derart abgeleitet werden, so dass die ausgewählte Anweisung (roter Knoten) erreicht wird und somit ausgeführt werden kann. Die Eingabeparameter der Funktion müssen so gewählt werden, dass alle Bedingungen entlang des Pfades (if, for, $while$ etc.) bis hin zur entsprechenden Anweisung gültig werden. Variablen X, die formale Parameterliste P_f und alle I/O Eingabedaten P_{io} des Steuerungsquelltextes werden direkt auf eine Menge von Variablen X des CSP abgebildet. Deren endlicher Wertebereich D_i kann über den korrespondierenden Datentyp, der üblicherweise *BOOL* oder *INTEGER* ist, abgeleitet werden. Die Constraints C des CSP sind die Bedingungen entlang des Pfades von v_s

bis zur erreichbaren Anweisung s_i (roter Knoten) innerhalb des Abhängigkeitsgraphen.

Bei der Generierung des CSP Problems müssen aus dem Kontrollflussgraphen des IEC 61131-3 Quelltextes die zur Erreichbarkeit notwendigen Bedingungen entlang des Pfades in Minion Constraints überführt werden. Neben den tatsächlichen Constraints zur Bestimmung der Testeingabedaten sind für die deklarative Problembeschreibung die folgenden drei Sektionen notwendig:

- ****VARIABLES**** Die dem CSP zugrunde gelegten Variablen X mit deren dazugehörigen Wertebereich D müssen in dieser Sektion deklariert werden. Zur Unterscheidung der Wertebereiche stehen im wesentlichen BOOL (0 oder 1), DISCRETE (diskreter Wertebereich) sowie BOUND und SPARSEBOUND (beliebige Menge an konkreten Werten) zur Verfügung.
- ****SEARCH**** Über diese Sektion können die zu bestimmenden Variablen spezifiziert werden. Des Weiteren wird über eine PRINT-Direktive die Reihenfolge der Wertebelegung aller Variablen festgelegt, so dass eine automatisierte Auswertung der CSP Lösung erfolgen kann.
- ****CONSTRAINTS**** Die Constraints entsprechen mathematischen Ungleichungen über alle problembezogenen Variablen und begrenzen dadurch den Lösungsraum. Die Formulierung der Constraints mit Hilfe Minion-kompatibler Anweisungen ist in der Tabelle 4.3 dargestellt. Dabei lassen sich atomare, gewichtete und verknüpfte Bedingungen in der Ausgangssprache unterscheiden und auf CSP Constraints abbilden.

Die für die Formulierung des Constraints relevanten Bedingungen des strukturierten Textes bestehen aus booleschen Ausdrücken aus der Menge der verfügbaren Variablen. Für die Testdatengenerierung ist die Bestimmung der Eingabedaten, d.h. der Wertebelegung von P_f und P_{io} entscheidend. Dennoch ist es geläufig, dass Bedingungen nicht ausschließlich unmittelbar aus den o.g. Parametern bestehen, sondern teilweise oder auch vollständig aus lokalen Variablen, deren Wertebelegung durch die Eingabeparameter bestimmt werden. Auf diese Relation über die Variablen X und deren korrespondierenden Wertebereich D kann über den transitiven Relationscharakter zurückgeschlossen werden. Hiermit werden die lokalen Variablen durch die Parameter P_f und P_{io} substituiert. Der Algorithmus 1 beschreibt den vollständigen Ablauf der Erzeugung der CSP Minion-Deklaration als Grundlage der Testdatengenerierung. Als Eingabe dienen der Kontrollflussgraph, die zu erreichende Anweisung im Quelltext des Steuerungsprogramms (ermittelt über I/O Mapping zu potentiellem Fehler), die Menge der Variablen und deren Definitionsbereich sowie die Eingabeparameter der Steuerung (I/O) als auch der Funktion. Mit Hilfe einer Tiefensuche werden alle Anweisungen vom Start bis zur erreichenden Anweisungen ermittelt. Anschließend werden diese in umgekehrter Reihenfolge überprüft, welche davon eine Bedingung (boolscher Ausdruck) enthält. Da Bedingungen häufig durch lokale Variablen ausgedrückt werden, müssen diese zu den eigentlichen Eingabeparametern X' expandiert werden. Daraus resultiert somit auch der tatsächliche Definitionsbereich D', der je nach verwendeten Datentypen kleiner sein kann als die eigentlichen Eingabedaten ermöglichen. Alle

Data : CFG, rS, X, D, P_f, P_{io}
Result : CSP Minion (C,X',D')
begin
  $C \leftarrow \emptyset$
  $X' \leftarrow \emptyset$
  $D' \leftarrow \emptyset$
  P(CFG, v_{rS})= $\{v_s, v_1, \ldots, v_{k-1}, v_k\}$ $\leftarrow$ DFS(CFG, rS);
  for $i \leftarrow (k-1)$ **to** 0 **do** // Bottom up Auswertung des CFG
    if v_i *has Condition* c_i **then**
      $X' \leftarrow TC(x_j \in c_i) \subseteq \{P_f \uplus P_{io}\}$
      $D' \leftarrow D_{min}(d_j \in c_i)$
      $C \leftarrow TC(c_i)$
    end
  end
end

Algorithmus 1 : Algorithmus zur Generierung des CSP Minion Problems

Restriktionen C ergeben sich aus der Gesamtheit aller Bedingungen entlang des Pfades bis zur zu erreichenden Anweisung.

Mit diesem Formalismus lassen sich Testdaten für SPS Applikationen generieren. Der Beispielquelltext in Listing 4.1 zeigt eine einfache Implementierung einer 90 Grad Rotationsfunktionalität einer Weiche. Diese wird dabei durch einen Motor gesteuert, der in zwei Richtungen bewegt werden kann. Ein zusätzlich angebrachter Sensor kann die Winkelposition der Weiche ermitteln. Wenn sich der final gemessene Winkel nicht zwischen 89 und 91 Grad befindet, wird der Motor betätigt, bis die Endstellung erreicht wird. Mit dem eingeführten Verfahren des CSP können Testdaten generiert werden, so dass Zeile 2 des Listings 4.1 erreicht und somit ausgeführt wird. Für komplexere, zyklisch ausgeführte Programmfragmente bedarf es der Erweiterung um die Datenabhängigkeit, da in diesem Fall manche Pfade erst nach einer mehrfachen Auswertung erreichbar werden und somit auch ein mehrstufiges Testgenerierungsverfahren notwendig macht.

Listing 4.1: Einfaches Rotationsbeispiel in ST zur Testdatengenerierung

```
1 01: IF sAngle > 88 AND sAngle < 92 THEN
2 02:     motor := 0;
3 03:     finished := true;
4 04: ELSE
5 05:     motor := 1;
6 06: END_IF
```

4.4.3 Testdatengenerierung auf Basis der Datenabhängigkeit

Mit dem Verfahren aus 4.4.2.2 kann der Quelltext entlang der Pfade hin zu der erreichbaren Anweisung analysiert und die korrespondierenden Testdaten ohne weitere

Analyse abgeleitet werden. In diesem Fall ist eine direkte Evaluation aller Eingabeparameter zur Generierung der Testeingabedaten ausreichend. Durch die zyklische Ausführungslogik von SPS Applikationen werden POEs mehrfach aufgerufen, bis ein entsprechender Zustand der Software, der mit einem physikalischen Zustand der Maschine oder Anlage bzw. dem Prozess übereinstimmt, erreicht wird.

Ein umfangreicheres Beispiel ist in Listing 4.2 gegeben. Es bildet einen Transportprozess unter Zuhilfenahme der vorher eingeführten Weiche von einem Startwinkel *fromPos* auf einen Zielwinkel *toPos* ab. Die Testdatengenerierung für Zeile 5 impliziert, dass *arrive*, *bottle* sowie *sAngle != toPos* wahr werden müssen. Die lokalen Variablen werden wie folgt initialisiert: *arrived(false)*, *finished(false)*, *destination(false)*.

Listing 4.2: Erweitertes Rotationsbeispiel

```
01: IF NOT arrived AND sAngle <> fromPos THEN
02:   motor := 1;
03: ELSIF arrived AND bottle THEN
04:   IF sAngle <> toPos THEN
05:     motor := -1;
06:   ELSE
07:     motor       := 0;
08:     destination := true;
09:   END_IF
10: ELSE
11:   motor    := 0;
12:   arrived := true;
13: END_IF
14: IF destination AND NOT bottle THEN
15:   finished := true;
16: END_IF
```

Die Variable *arrived* muss wahr werden, so dass das Testdatengenerierungsverfahren in den entsprechenden Zweig (roter Pfad in Abbildung 4.15) absteigen kann. Da *arrived* mit falsch initialisiert wird, kann diese Bedingung bei einmaliger Ausführung niemals wahr werden. Bedingt durch die zyklische Ausführung wird *arrived* in Zeile 12 zu wahr modifiziert. Diese Zuweisung kann nur dann erreicht werden, wenn die Bedingungen in Zeile 01 und 03 falsch sind. Die def-use Datenabhängigkeit der Variablen *arrived* zwischen den Zeilen 03 und 12 (grüner Pfad in Abbildung 4.15) deckt den Einfluss des Kontrollflusses in Zeile 03 auf (gestrichelter grüner Pfad in Abbildung 4.15). Folglich kann Zeile 05 mit dem in Kapitel 4.4.2.2 eingeführten Konzept genau dann erreicht werden, wenn ein vorheriger zyklischer Aufruf Zeile 12 durchlaufen hat, so dass *arrived* wahr wurde. Folglich muss eine auf Basis der Abhängigkeit vorgelagerte Bewertung des Quelltextes erfolgen, so dass die relevanten Testdaten zur Erreichbarkeit der geforderten Anweisung bei einer erneuten Auswertung (Lösung des CSPs) generiert werden können.

4.4.3.1 Konstruktion des Abhängigkeitsgraphen

Damit ein bestimmter Zweig bzw. die geforderte Anweisung erreicht und ausgeführt werden kann, muss der Abhängigkeitsgraph mehrfach evaluiert werden. Dabei entspricht jeder Durchlauf einem zeitdiskreten Parametersatz, der den anschließenden Testfällen konsekutiv zugrunde gelegt werden. Die Anzahl der Evaluierungsdurchläufe hängt von den Datenabhängigkeiten zwischen Knoten disjunkter Zweige ab. Um diese Beziehungen in den Lösungsansatz aus 4.4.2.2 integrieren zu können, muss der Abhängig-keitsgraph (Definition 11) um die Datenabhängigkeit zwischen Kontrollvariablen zu einem Multigraphen erweitert werden, siehe Definition 12.

Definition 12 (*Erweiterter Abhängigkeitsgraph (def-use)*)

Ein erweiterter Abhängigkeitsgraph $G'(V,\{E,F\},v_s,v_e)$ besteht aus einer Menge von Knoten V und gerichteten Kanten $E : V \times V$, die den Kontrollfluss zwischen Quelltextelementen repräsentieren und gerichteten Kanten $F_{w,e} : V \times V$ für eine def-use Datenabhängigkeit zwischen der Definition einer Kontrollvariablen $w \notin \{P_f \uplus P_{io}\}$ und deren Verwendung in einem Quelltextelement e. Ein vollständiger erweiterter def-use Abhängigkeitsgraph existiert genau dann, wenn für alle Kontrollvariablen $W \subset V$ eine Kante zu abhängigen Codeelementen $E \subset V$ existiert. Im einfachsten Fall entspricht ein Codeelement einer Variablen.

Data : $G(V,E,v_s,v_e)$, W
Result : $G'(V,\{E,F\},v_s,v_e)$
$F \leftarrow \emptyset$
for $w \in W$ **do**
  if $\exists(w,e) \in V \times V; w \in v_i, e \in v_j$ *mit* $i \neq j$ **then** // Shunting-Yard
    $F \leftarrow (v_i,v_j)$
  end
end

Algorithmus 2 : Algorithmus zur Ermittlung der def-use Datenabhängigkeit

Der Abhängigkeitsgraph G' erweitert den Basisgraphen G durch eine Bottom-up Analyse. Neben den kontrollflussbasierten Relationen zwischen den Knoten, wird der Graph um die def-use Beziehung erweitert. Die Definition (def) einer Variablen besteht aus einer Zuweisung im zugrundegelegten Quelltext, d.h. einer Variablen wird ein entsprechender Ausdruck zugewiesen und dadurch definiert. Der Wert dieser Variablen kann an einer oder an mehreren Stellen verwendet (use) werden und zwar in Form von weiteren Zuweisungen auf zusätzliche Variablen oder in Bedingungen zur Pfadmanipulation. Lokale Variablen bieten zusätzliche Kontrollmechanismen, die aufgrund des Systemzustandes gesetzt werden, und sind deshalb meist Bestandteil einer Bedingungsprüfung. Durch die zyklische Ausführungslogik der Softwarebausteine müssen derartige lokale Variablen nicht entlang des direkten Pfades definiert bzw. manipuliert werden, sondern können auch in einem disjunkten Pfad existieren. Die Auswertung der Datenabhängigkeit liegt der erweiterten Testdatengenerierung in Kapitel 4.4.3.2 zugrunde. Der Algorithmus 2 ermittelt die auf eine beliebige Bedingung innerhalb des Kontrollflussgraphen G existierende def-use Abhängigkeit und konstruiert

den erweiterten Abhängigkeitsgraphen G'. Bei der Konstruktion von G' dient G als Grundlage, der mit der Menge der verwendeten Variablen W durchsucht wird. Wenn eine Kontrollvariable $w \in W$ Teil eines Quelltextelements e ist, so wird jene Kante (v_i,v_j) zu F hinzugefügt, für die e Bestandteil des Ausdrucks in Knoten v_i und w Bestandteil der Bedingung in Knoten v_j ist. Die resultierende Relation F wird dem Abhängigkeitsgraphen G zu G' hinzugefügt. Zur Bewertung der Codeelemente e müssen die Ausdrücke (right value) mit dem Parser analysiert und dadurch Variablen, Operatoren und Aufrufe separiert werden. Die Transformation der Infix Notation in eine analysierbare Postfix Notation erfolgt nach dem Shunting Yard Algorithmus von Dijkstra [Dij61].

4.4.3.2 Erweiterte Testdatengenerierung bei Datenabhängigkeiten

Der in Kapitel 4.4.2 eingeführte Formalismus zur Testdatengenerierung erstellt aus dem ursprünglichen Softwarebaustein den Kontrollflussgraphen, der als Grundlage für die Generierung des Constraint Satisfaction Problems dient. Die Lösung korrespondiert mit der minimalen Menge an Parameterwerten zur Erreichbarkeit einer geforderten Anweisung im ursprünglichen Softwarebaustein. Durch die zyklische Ausführungslogik ist eine Anweisung im Regelfall nicht ausschließlich durch die einmalige Stimulation des Testobjekts über die formalen Eingabeparameter erreichbar. Häufig führen Fallunterscheidungen, die durch Hilfsvariablen in Form von lokalen Variablen und Berechnungskombinationen mit weiteren Funktionsbausteinen den Kontrollfluss manipulieren, zu Anweisungen, die den Systemzustand der Steuerungssoftware manipulieren, so dass nachgelagerte Anweisungen erst erreichbar werden. Eine vorgegebene Anweisung kann genau dann erreicht werden, wenn alle zu dieser Anweisung führenden Pfadbedingungen wahr werden. Die Variablenwerte in Pfadbedingungen sind meist abhängig von Neuzuweisungen in parallelen Pfaden, deren Ausführung vorab erfolgen muss.

Der zyklischen Ausführungslogik geschuldet, bedarf es einer mehrfachen Stimulation des Testobjekts. Dadurch können dessen interne Zustände derart variiert werden, dass nach einer konsekutiven, zeitdiskreten Stimulation die intendierte Anweisung erreicht werden kann. Der im vorangegangen Kapitel eingeführte erweiterte Abhängigkeitsgraph wird über den Algorithmus 2 bestimmt, und ist die Voraussetzung zur Lösung der Testdatengenerierung für zyklisch ausgeführte Softwarebausteine.

Die Erreichbarkeit der Anweisung in Zeile 5 des in Listing 4.2 aufgeführten Rotationsbeispiels ist abhängig von der Belegung der lokalen Variablen *arrived*, die als Kontrollvariable den internen Zustand repräsentiert. Abbildung 4.15 zeigt den relevanten Ausschnitt aus dem Kontrollflussgraphen G, worin die roten Transitionen den Pfad zur Testdatengenerierung darstellen. Der grüne Pfad erweitert G zum defuse Abhängigkeitsgraphen G'. Die Wertebelegung der Variablen *arrived* in Knoten 3 bestimmt die Erreichbarkeit des Knotens 5. In Knoten 12 (def) erfolgt eine Zuweisung der lokalen Variablen *arrived*, die die Pfadänderung in Knoten 3 (use) zur Erreichbarkeitserfüllung von Knoten 5 realisiert.

Der gesamte Ablauf der Testdatengenerierung für zyklische Funktionsbausteine ist in Algorithmus 3 beschrieben. Zur Berechnung der Testeingabedaten $T_s(F)$ werden dem Algorithmus der Softwarebaustein F und die zu erreichende Anweisung s als

Parameterdaten übergeben. Der erste Transformationsschritt überführt den gegebenen Softwarebaustein F gemäß den Vorschriften in Tabelle 4.2 in den Kontrollflussgraphen G. Der Pfad zur Anweisung s in G wird über eine Tiefensuche DFS realisiert und liefert die Knoten- und Transitionsabfolge zur Traversierung des Kontrollflussgraphen. Mit dem Algorithmus 1 wird das initiale Constraint Satisfaction Problem CSP_{t_0} für den Pfad P durch G abgeleitet. Bei Pfaden, die vollständig durch die Stimulation der formalen Eingabeparameter erreichbar sind, d.h. keine lokalen Variablen enthalten, kann durch die Lösung dieses CSPs der Testeingabedatensatz $T_s(F)$ mit Einsatz des Constraint Solvers Minion ermittelt werden. Für alle entlang des Pfades auftretenden Bedingungen müssen die darin enthaltenen lokalen Variablen X_L der Signatur von F extrahiert werden. Für alle existierenden lokalen Variablen muss mit Algorithmus 2 die def-use Datenabhängigkeit in G zu G' errechnet werden, um die lokalen Variablen so zu expandieren, dass alle Bedingungen entlang des Pfades durch Eingabeparameter des Testobjekts beschrieben werden können. Anschließend muss der Pfad P zur Erreichbarkeit der Anweisung $s(x)$, die die lokale Variable x definiert, ermittelt werden. Anschließend lässt sich für den neu ermittelten Pfad ein zusätzliches CSP_{t_i} in G' erstellen.

Die Generierung der Eingangsdaten erfolgt in Form der Lösung mehrstufiger Constraint Satisfaction Problems, bei denen jede Lösung den Testeingabedaten zu einem bestimmten diskreten Zeitpunkt t_i entspricht. Die Eingabedaten zur Stimulation müssen in umgekehrter Reihenfolge zum korrespondierenden CSP aufgestellt und abgeleitet werden. Bei k zu expandierenden, abhängigen, lokalen Variablen entstehen die $k+1$ CSPs t_0,t_1,...,t_k, die von k bis 0 mit Hilfe des Constraint Solvers Minion gelöst werden und deren Lösung die finalen Testeingabesequenzen darstellen. Die Lösung des k-ten CSPs entspricht dabei dem ersten zeitdiskreten Eingabedatensatz, das

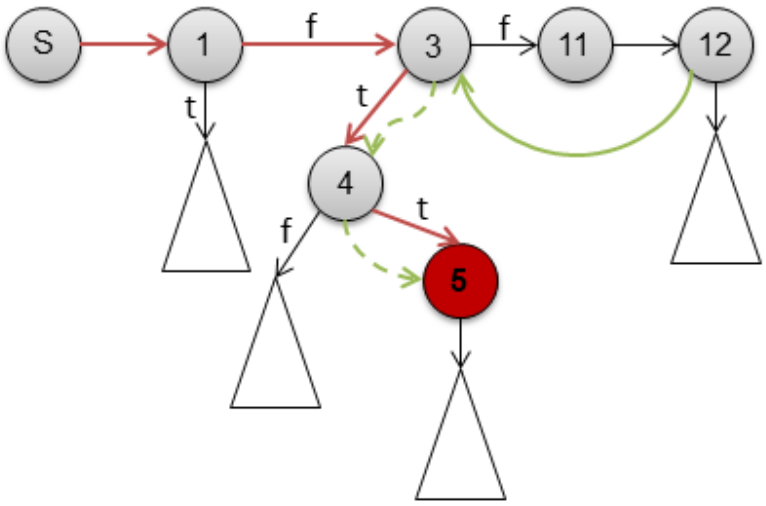


Abb. 4.15: Datenabhängigkeitsgraph zur Testdatengenerierung für Zeile 5 in Listing 4.2. Rot symbolisiert den Pfad zur Testdatengenerierung für die Anweisung in Zeile 5. Die grüne Linie zeigt die Datenabhängigkeit der Variablen *arrived*, die gestrichelte Linie zeigt den Pfad in Bezug zu *arrived*.

Data : Code F, Statement s
Result : Test Input Data $T_s(F)$
$CSP \leftarrow \emptyset$
$T_s(F) \leftarrow \emptyset$
$G = \text{transform}(F)$; // Tabelle 4.2
$P = \text{find}(G, s)$; // DFS
$i = 0$;
$CSP_{t_i} \leftarrow \text{derive}(P \text{ of } G)$ // Algorithmus 1
while P *contains local Variables* X_L **do**
  $inc(i)$;
  $G' = \forall x \in X_L, \text{extend}(G)$; // Algorithmus 2
  $P = \text{find}(G', s(x))$; // DFS
  $CSP_{t_i} \leftarrow \text{derive}(P \text{ of } G')$ // Algorithmus 1
end
for $i = \#CSP$ **to** 0 **do**
  $T_s(F) \leftarrow \text{solve}(CSP_{t_i})$ // Constraint Solver
end

Algorithmus 3 : Gesamtalgorithmus zur vollständigen Testdatengenerierung zyklischer Funktionsbausteine

$(k-1)$-te dem zweiten Testeingabedatensatz bis zum 0-ten CSP, das die Constraints zur Testdatengenerierung der Anweisung s entlang des ermittelten Pfades P in G repräsentiert. Der gesamte Ablauf ist im Algorithmus 3 beschrieben.

4.4.4 Relationen zwischen Sensor Aktor Funktionseinheiten

Bei der Funktionserfüllung von Maschinen und Anlagen werden Sensoren und Aktoren geeignet kombiniert. Dadurch ergeben sich funktionale Querbezüge bzw. Abhängigkeiten für kleine funktionale Einheiten. Bei der mechatronischen Modularisierung von Maschinen und Anlagen werden häufig sog. Sensor-Aktoren-Listen herangezogen, da diese den funktionalen Zusammenhang beschreiben und dadurch die Schnittstelle definiert und anschließende Komposition ermöglicht wird.

Bei der Überprüfung von Fehlerszenarien während des Tests ergeben sich durch die Berücksichtigung der Sensor Aktor Relation weitere Kombinationsmöglichkeiten in den Parametereinstellungen der Simulation. Durch das bisher beschriebene Verfahren können Testeingabedaten für Defekte in Komponenten, deren Ansprechverhalten nicht abhängig von Querbezügen ist, unmittelbar ermittelt werden. Im Maschinen- und Anlagenbau können die Relation zwischen Sensoren und Aktoren häufig auf einfache funktionale Beziehungen reduziert werden, wie in Abbildung 4.16 dargestellt.

Das Wissen über diese (meist einfachen) Relationen muss dem Generierungsverfahren zugeführt werden. Dies kann in Form von Tabellen realisiert werden, die die Zusammenhangsinformationen zwischen dem betroffenen Sensor und Aktor beschreiben. Dadurch können unter Berücksichtigung der zyklischen Ausführung auch funktionale Zusammenhänge aufgelöst werden. Aus den berechneten Parametern (I/O-Wertebelegung) lassen sich Optionen der Simulationszustandskonfiguration ermitteln.

Des Weiteren können auch Relationen zwischen mehreren Sensoren berücksichtigt werden, die eine gemeinsame funktionale Einheit darstellen.

4.4.5 Äquivalenzklassenbildung der Testeingabedaten

Die durch den Algorithmus 3 ermittelten Testeingabedaten $T_S(F)$ werden durch gültige Wertepaare über alle Variablen ausgedrückt, d.h. Wertekombinationen, die alle Einschränkungen des Constraint Satisfaction Problems erfüllen. Der Umfang der generierten Werte hängt maßgeblich von dem ausgewählten Fehler und dem dazugehörigen Steuerungsquelltext ab. Dabei beschreiben die Lösungen stets eine Untermenge des Kreuzproduktes der Wertebereiche über alle Variablen, d.h. Eingabeparameter, wodurch eine sehr hohe Ergebnistiefe erreicht werden kann. Bei der vollständigen Berechnung gültiger Wertepaare ergibt sich somit eine sehr hohe Anzahl von Testeingabedaten zum Erreichen der entsprechenden Anweisung. Für n Parameter und m errechneten Lösungen setzen sich die Testeingabedaten $T_S(F)$ wie folgt zusammen:

$$T_S(F) = \begin{pmatrix} t_{11} & t_{21} & \dots & t_{m1} \\ t_{12} & t_{22} & \dots & t_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ t_{1n} & t_{2n} & \dots & t_{mn} \end{pmatrix} \tag{4.3}$$

Jede Spalte repräsentiert einen gültigen Lösungsvektor $T_i = (t_{i1}, t_{i2}, \dots, t_{in})$. Neben der Erhöhung der Aussagekraft der Testeingabedaten durch die starke Bindung an mechanische, permanente Fehler, kann die Aussagekraft der ermittelten konkreten Werte weiter erhöht werden, in dem deren Ähnlichkeit untersucht und Äquivalenzklassen gebildet werden. Die Äquivalenzklassen des Wertebereichs aller Parameter formen eine Äquivalenzklassengruppe E_g, die wie folgt definiert ist:

$$E_g = \begin{pmatrix} t_{i1} \\ \vdots \\ t_{j1} \end{pmatrix} \times \begin{pmatrix} t_{k2} \\ \vdots \\ t_{l2} \end{pmatrix} \times \dots \times \begin{pmatrix} t_{rn} \\ \vdots \\ t_{sn} \end{pmatrix} \tag{4.4}$$

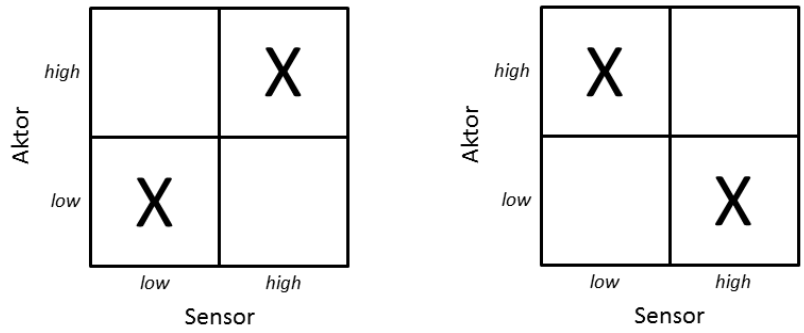


Abb. 4.16: Typische Relationen zwischen Sensor und Aktor

Für alle m Lösungen können $1 \ldots m$ Äquivalenzklassengruppen ermittelt werden. Jede Äquivalenzklasse stellt dabei eine Untermenge des Wertebereichs des korrespondierenden Eingabeparameters dar. Zur Zuweisung von Lösungen zu Äquivalenzklassen muss gelten:

$$\forall(T_i, T_j) \subset T_S(F); \Delta(T_i,T_j) = 1; i \neq j \tag{4.5}$$

Damit Vektoren zu einer gemeinsamen Äquivalenzklasse zusammengeführt werden, müssen sie sich paarweise in genau einem Wert unterscheiden. Dies wird über die folgende Beziehung ermittelt:

$$\Delta : \sum_{k=1}^{|T_i|} \delta(t_{ik},t_{jk}) \tag{4.6}$$

Der Abstand zweier Lösungen wird mit einer Δ-Funktion über den Vergleich aller Wertepaare der gültigen Lösungen bestimmt. Dazu wird der Abstand aller Elemente wie folgt ermittelt:

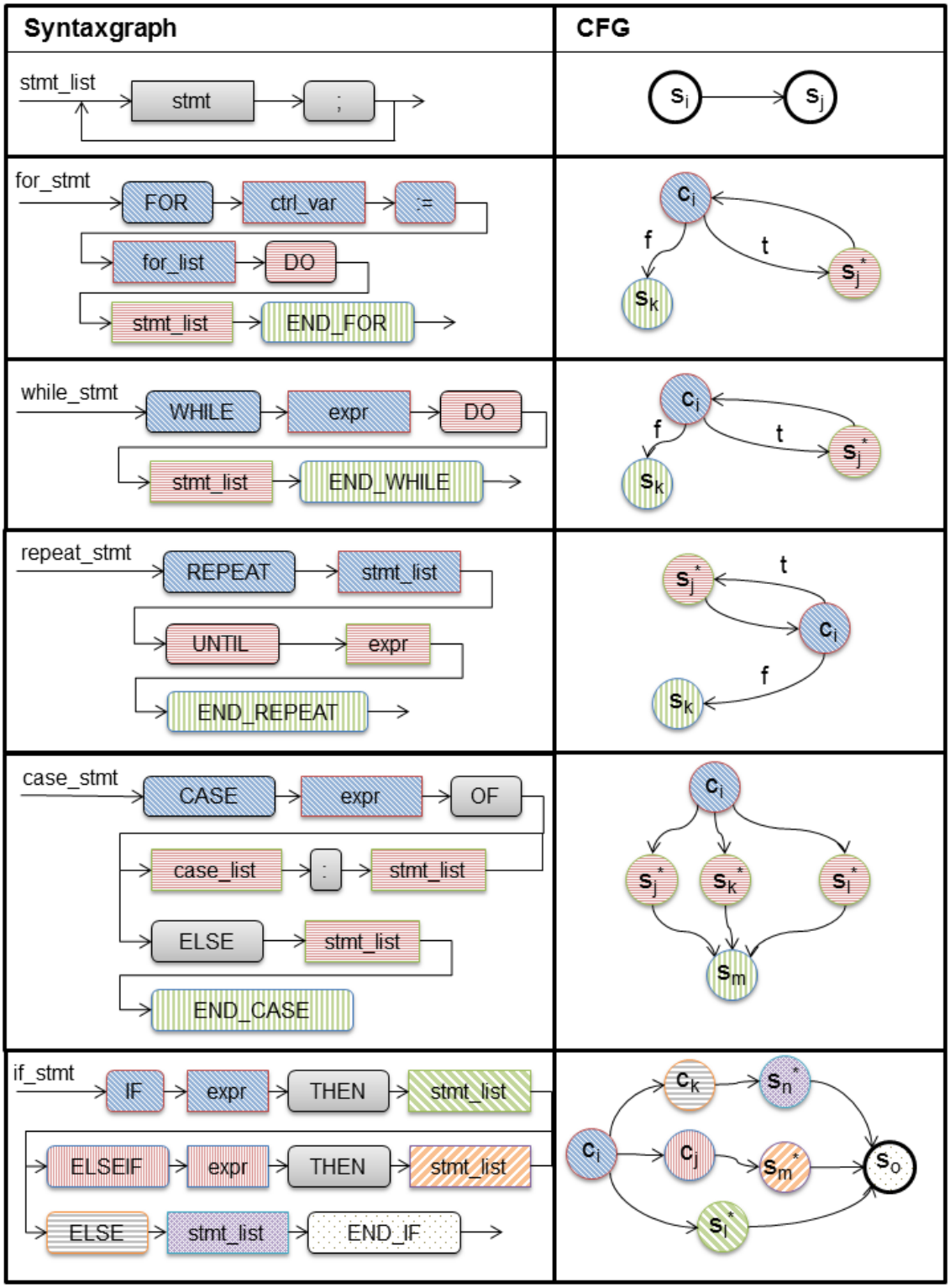
$$\delta(t_i,t_j) = \begin{cases} 1 & \text{for } t_i \neq t_j \\ 0 & \text{for } t_i = t_j \end{cases} \tag{4.7}$$

Durch die Anwendung des Kreuzproduktes über alle Äquivalenzklassen lassen sich alle gültigen Lösungen (Testeingabedaten) in $T_S(F)$ erzeugen. Die Mächtigkeit der Äquivalenzklassen und die Anzahl der Gruppen hängen stark von der zu analysierenden IEC 61131-3 Steuerungsapplikation und dem modellierten Fehler ab. Je mächtiger die Äquivalenzklasse, d.h. je mehr Parameterwerte zugewiesen werden können, desto aussagekräftiger (von hoher Bedeutung) ist der gesamte Parametersatz der Äquivalenzklassengruppe. Durch die quantitative Bündelung der Wertekombinationen kann durch die Auswahl jeweils eines Repräsentanten der Parameter die Aussagekraft zusätzlich erhöht werden.

Beispiel: In dem folgenden Beispiel sind drei Äquivalenzgruppen E_{g1}, E_{g2}, E_{g3} aufgeführt, die jeweils drei Parameter beinhalten. Die Gruppe E_{g1} hat ein höheres Gewicht, da diese mehrere Parameterkombinationen vereint. Dies bedeutet, dass durch jede der Wertekombinationen derselbe Pfad der Steuerungssoftware durchlaufen wird.

$$E_{g1} = \begin{pmatrix} \{0,1\} \\ \{0,90,180\} \\ \{0,1,2\} \end{pmatrix}, \qquad E_{g2} = \begin{pmatrix} \{0\} \\ \{90\} \\ \{3\} \end{pmatrix}, \qquad E_{g3} = \begin{pmatrix} \{1\} \\ \{270\} \\ \{2\} \end{pmatrix} \tag{4.8}$$

Tab. 4.2: Abbildung der ST Kontrollstrukturen auf den Kontrollflussgraphen, c_i und s_i sind Knotenelemente, s_i^* ist eine beliebige Anzahl von Anweisungen



ST-Ausdruck		**Minion Constraint**
Atomar	$a < b$	ineq(a, b, -1)
	$a > b$	ineq(b, a, -1)
	$a = b$	eq(a, b)
	$a \neq b$	diseq(a, b)
	$a \leq b$	ineq(a, b, 0)
	$a \geq b$	ineq(b, a, 0)
Gewichtet	$c_1 \cdot a + c_2 \cdot b \leq c$	weightedsumgeq([c_1,c_2], [a,b], c)
	$c_1 \cdot a + c_2 \cdot b \geq c$	weightedsumleq([c_1,c_2], [a,b], c)
Verknüpft	NOT a	eq(a, 0)
	A AND B	watched-and(A, B)
	A OR B	watched-or(A, B)
Zuweisung	a := b	eq(a, b)
	a := b + c	ineq(a, b, c) ineq(b, a, -c)
	a := b * c	product(b, c, a)

Tab. 4.3: Transformation atomarer, gewichteter und verknüpfter ST-Bedingungen und Zuweisungen in Minion Constraints

4.5 Testdatengenerierung durch symbolische Ausführung

Die symbolische Ausführung von Programmen überführt Ausdrücke im Quelltext durch mathematische Repräsentanten, wodurch formale Pfadanalysen ermöglicht werden. In diesem Forschungsfeld werden durch die vorhandenen, etablierten Werkzeuge meist C Programme analysiert, da diese die höchste Verbreitung[1] haben.

Ein weiteres wesentliches Unterscheidungsmerkmal der beiden Programmierwelten stellt die zyklische Ausführungslogik der IEC 61131-3 basierten Applikationen im Vergleich zu C Programmen dar. Aufgrund dieser charakteristischen Eigenschaft sind manche Pfade nur nach mehrfacher Ausführung, d.h. Stimulation erreichbar.

4.5.1 Das Werkzeug KLEE

KLEE [CDE08] ist einer der prominentesten Vertreter für symbolische Ausführung von C Programmen. In Abbildung 4.17 ist die Architektur von KLEE dargestellt. Dabei wird die LLVM Compiler Infrastructure [Lat02] verwendet, um den Ausgangsquelltext zu instrumentieren und in einen analysierbaren Zwischencode zu überführen. In der symbolischen Umgebung werden die extrahierten Variablen einem Constraint Solver übergeben, der durch Optimierungen wie Constraint Elimination, Caching und Environment Modeling die hohe Performance von KLEE ermöglicht. Die Ausgabe aller Durchläufe von KLEE (Permutation der Eingabedaten) erfolgt in einem KLEE internen Format.

Die Werkzeugkette KLEE wird mit den Kommandos aus Listing 4.3 bedient.

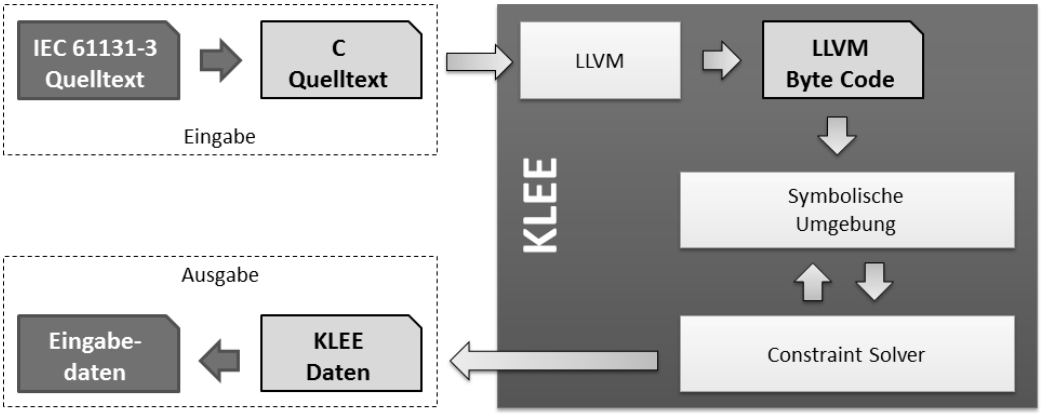


Abb. 4.17: Die Architektur der KLEE Integration

1 Die Programmiersprache C stellt gemäß TIOBE Index (`http://www.tiobe.com`) eine der am stärksten verbreiteten Sprachen dar (Stand: 2016).

Listing 4.3: Die KLEE Werkzeugkette

```
# C-Programm in LLVM Byte Code überführen
$ llvm-gcc --emit-llvm -c -g programm.c
# Byte Code analysieren und Eingabedaten generieren
$ klee programm.o
# Ergebnis des ersten Pfades ausgeben
$ ktest-tool --write-ints klee-last/test000001.ktext
# Ergebnis des zweiten Pfades ausgeben
$ ktest-tool --write-ints klee-last/test000002.ktext
```

4.5.2 Transformation von IEC 61131-3 in C für KLEE

Für die Verwendung von KLEE für IEC 61131-3 (ST) Programme bedarf es der Transformation dieser Programme in C Quelltext. Dazu müssen die Datentypen, Kontrollstrukturen sowie Anweisungen und Ausdrücke aus ST in C unter Wahrung der Semantik überführt werden. Die syntaktischen Unterschiede der beiden Sprachwelten werden in den folgenden drei Unterabschnitten näher erläutert.

4.5.2.1 Datentypen

Im Kopf eines Funktionsbausteins erfolgt die Deklaration der Variablen mit ihren entsprechenden primitiven Datentypen. In der Programmiersprache C müssen ebenfalls zunächst alle Variablen deklariert werden, bevor diese verwendet werden können[1]. Bei der Deklaration gibt es einen syntaktischen Unterschied zwischen den beiden Sprachen. So zeigt Listing 4.4 die Deklaration einer Ganzzahl mit und ohne Wertzuweisung in ST und Listing 4.5 die dazugehörige Syntax in C. Die Abbildung der Datentypen zwischen den beiden Sprachen ist in Tabelle 4.4 dargestellt. Datei stellen die arithmetischen Basisdatentypen keine Besonderheit dar, wohingegen es keinen direkten boolschen Datentypen in C gibt und auch Zeichenketten nicht unmittelbar vorhanden sind. Speziell die Behandlung von Zeichenketten unterscheiden sich in den beiden Sprachräumen und muss in C anders behandelt werden, da bspw. die Prüfung auf Gleichheit nicht mit dem Gleichheitsoperator erfolgen kann.

Listing 4.4: Variablendeklaration in ST

```
1 Var1 : INT;
2 Var2 : INT := 5;
```

Listing 4.5: Variablendeklaration in C

```
1 int Var1;
2 int Var2 = 5;
```

1 Häufig erlauben moderne Compiler trotzdem eine lazy evaluation. Der GNU C Compiler kann über das Flag -pedantic zur strikten Standardkonformität gezwungen werden.

Tab. 4.4: Datentypenabbildung von ST nach C

Datentyp	Typ ST	Typ C
Integer	INT	int
Gleitkomma	DOUBLE	double
Gleitkomma	LREAL	double
Boolsch	BOOL	char
Zeichenkette	STRING	char*

4.5.2.2 Kontrollstrukturen

Für die Erreichbarkeitsanalyse eines Programms sind die Bedingungen von zentraler Bedeutung, da diese die Aufgabe der Pfadmanipulation übernehmen. In Analogie zur expliziten Betrachtung der Datenabhängigkeit in Abschnitt 4.4 müssen die Kontrollstrukturen aus dem Ausgangsquelltext (ST) semantisch eindeutig interpretiert und im Falle der Transformation in C Quelltext für KLEE auch syntaktisch korrekt transformiert werden. In Tabelle 4.5 sind die betrachteten Kontrollstrukturen der Sprachen ST (IEC 61131-3) und C in der jeweiligen Syntax gegenüber gestellt. In der Tabelle sind nur die jeweils relevanten Non-Terminal Symbole angegeben. Die vollständige Expansion ist der jeweiligen Spezifikation zu entnehmen [ISO11], [IEC03a].

Ein wesentlicher Unterschied zwischen den beiden Sprachen sind die verwendeten Schlüsselwörter zur Kennzeichnung von Blöcken. Die Programmiersprache ST ist an Pascal angelehnt und verwendet im wesentlichen **BEGIN** und **END** zum Öffnen und Beenden eines Blocks. In der Sprache C werden dazu die geschweiften Klammern **{** und **}** verwendet. Im Syntaxvergleich sind die geschweiften Klammern bei C deshalb nicht aufgeführt, weil diese bei der weiteren Expansion des Non-Terminal Symbols statement mit enthalten sind, sofern es sich um ein mehrzeiliges Statement handelt. Ein Transformationsbeispiel für switch-case ist in Listing 4.6 und Listing 4.7 dargestellt.

Listing 4.6: Switch-Case Beispiel in ST

```
1 CASE IX OF
2   0:   QX := 1;
3   90:  QX := 3;
4   180: QX := 2;
5   270: QX := 0;
6 ELSE
7   QX := 0;
8 END_CASE
```

Tab. 4.5: Transformationsübersicht ST nach C

Kontrollstruktur	**Syntax ST**	**Syntax C**
IF	'IF' expr 'THEN' stmt_list 'ELSE' stmt_list 'END_IF'	'if' '(' condition ')' statement 'else' statement
CASE	'CASE' expr 'OF' case_list ':' stmt_list 'ELSE' stmt_list 'END_CASE'	'switch' '(' condition ')' statement
FOR	'FOR' ctrl_var ':=' for_list 'DO' stmt_list 'END_FOR'	'for' '(' for_init_statement [condition] ';' [expression] ')' statement
WHILE	'WHILE' expr 'DO' stmt_list 'END_WHILE'	'while' '(' condition ')' statement
DO WHILE	'REPEAT' stmt_list 'UNTIL' expr 'END_REPEAT'	'do' statement 'while' '(' expression ')' ';'

Listing 4.7: Switch-Case Beispiel in C

```
1  switch (IX)
2  {
3   case 0:
4    QX = 1;
5    break;
6   case 90:
7    QX = 3;
8    break;
9   case 180:
10   QX = 2;
11   break;
12  case 270:
13   QX = 0;
14   break;
15  default:
16   QX = 0;
17   break;
18 }
```

4.5.2.3 Anweisungen und Ausdrücke

Die wesentliche Funktionalität eines Programmes wird neben den Pfadunterscheidungen durch Anweisungen und Ausdrücke realisiert. Diese setzen sich aus Operatoren zusammen, die syntaktisch in den jeweiligen Sprachen im Detail differieren. In den Tabellen 4.6, 4.7 und 4.8 sind die betrachteten Kategorien von Operatoren in den jeweiligen Sprachen aufgeführt.

Tab. 4.6: Übersicht der mathematischen Operatoren in ST und C

Operator	**ST**	**C**
Zuweisung	':='	'='
Addition	'+'	'+'
Subtraktion	'-'	'-'
Multiplikation	'*'	'*'
Division	'/'	'/'
Modulo	'MOD'	'%'
Potenz	'**'	'pow()'

Zeiger, Zeigerarithmetik und die rekursive Auswertung weiterer aufgerufener Funktionen werden in dieser Arbeit nicht betrachtet.

4.5.3 Zusätzliche Anforderung der IEC 61131-3 Programme

Werkzeuge zur symbolischen Ausführung wie KLEE führen den zu untersuchenden Quelltext nur einmalig aus, d.h. Programmfragmente, die ein Gedächtnis besitzen (Status) und wie bei IEC 61131-3 basierten Programmen mehrfach hintereinander

Tab. 4.7: Übersicht der Vergleichsoperatoren in ST und C

Operator	**ST**	**C**
Gleichheit	'='	'=='
Ungleichheit	'<>'	'!='
Kleiner (gleich)	'<' ('<=')	'<' ('<=')
Größer (gleich)	'>' ('>=')	'>' ('>=')
Vergleich Zeichenkette	'='	'strcmp() == 0'

Tab. 4.8: Übersicht der logischen Operatoren in ST und C

Operator	**ST**	**C**
Logisch und	'AND'	'&&'
Logisch oder	'OR'	'||'
Logisch nicht	'NOT'	'!'

(zyklisch) ausgeführt werden, können somit nicht vollständig analysiert werden. Diese sog. Datenabhängigkeit ist in Abschnitt 4.4 explizit berücksichtigt.

4.5.3.1 Datenabhängigkeit als Transformationsregel

Zur Nachbildung einer oben beschriebenen Umgebung müssen die in C-Quelltext transformierten IEC 61131-3 Programme mehrfach hintereinander ausgeführt werden. Die Verwendung einer while Schleife ohne Endkriterium kann dazu führen, dass KLEE nicht mehr terminiert. Aus diesem Grund wird die zu analysierende Programmfunktion innerhalb einer Schleife mit endlicher Obergrenze aufgerufen, wodurch die zyklische Ausführung simuliert wird. Das dabei auftretende Problem, die richtige Obergrenze zu finden, lässt sich allgemein nicht lösen. Je nach Applikationsbezug muss dieser Wert nach oben angepasst werden.

Zur Generierung von Eingabedaten des C Programms müssen die lokalen Variablen eines Funktionsblocks entweder als static Variablen im transformierten C-Quelltext deklariert werden. Ein einfaches Beispiel zur Verdeutlichung ist in Listing 4.8 aufgeführt. Die zyklische Ausführungslogik wird bei der Transformation in C durch ein Ausrollen der Aufrufe nachgebildet. Dabei entspricht jeder Aufruf einem Zykluswechsel und somit einer Testobjektstimulation bei einem möglicherweise geänderten, internen Zustand. Dieser Schritt ist im Rahmen einer Transformationsvorschrift automatisierbar.

Diesem KLEE basierten Verfahren wird zugrunde gelegt, dass Eingabedaten genau dann berechnet wurden, wenn die eingefügte Assertion ausgeführt wurde. Bei zyklischen Abhängigkeiten ist die Häufigkeit des Funktionsaufrufs, bis die Assertion erreicht wird, vorab nicht bekannt. Es wäre denkbar, die Anzahl der Aufrufe auf einen bestimmten, hohen Wert (bspw. 100) zu setzen. KLEE wird anschließend bei der Berechnung die Assertion erreichen und die dazugehörigen Parameterdaten ermitteln. Dieser Ansatz hat jedoch zwei wesentliche Nachteile:

- Die Berechnungszeit wird durch die zusätzlichen Funktionsaufrufe und Berechnungsschritte unnötig erhöht.

- KLEE ermittelt die Eingabeparameter derart, dass erst mit Ausführung des letzten Funktionsaufrufs die angegebene Assertion ausgeführt wird, was somit auch zum Abbruch der Berechnung führt. Aus den berechneten Eingabedaten kann demnach auch nicht ermittelt werden, nach wie vielen Zyklen die Assertion tatsächlich erreicht worden wäre.

Um diese Tatsache zu umgehen, kann die Berechnung mit KLEE sukzessive um einen Funktionsaufruf erweitert werden, bis die Assertion erreicht wird. Dadurch wird die Anzahl der abhängigen Pfade so lange schrittweise erhöht, bis die richtige Anzahl gefunden wurde. In dem in Abschnitt 4.4 beschriebenen Verfahren wird bereits diese Abhängigkeit in einem Analyseschritt als Vorbereitung für die Formulierung des CSPs ermittelt. Diese Information wird für das Verfahren mit KLEE verwendet, so dass die Anzahl der notwendigen Funktionsaufrufe (Zyklen) unmittelbar integriert wird.

Listing 4.8: Zyklische Ausführungslogik durch Ausrollen der Aufrufe

```
1  #include <klee/klee.h>
2
3  int QX;
4
5  void func( int a, int b )
6  {
7      static int c = 0;
8      if ( a == 0 && b > 5 )
9          c = 1;
10     else if ( c == 1 )
11     {
12         QX = 0;
13         klee_assert( 0 );
14     }
15 }
16
17 int main()
18 {
19     int a1, a2;
20     int b1, b2;
21
22     // first cycle
23     klee_make_symbolic( &a1, sizeof( int ), "a1" );
24     klee_make_symbolic( &b1, sizeof( int ), "b1" );
25     func( a1, b1 );
26
27     // second cycle
28     klee_make_symbolic( &a2, sizeof( int ), "a2" );
29     klee_make_symbolic( &b2, sizeof( int ), "b2" );
30     func( a2, b2 );
31
32     return 0;
33 }
```

Das Ergebnis der symbolischen Ausführung enthält nun die zeitdiskrete Belegung der Eingabedaten, wobei während der Transformation eine beliebige Anzahl an Zyklen angenommen wurde. Demnach kann aus dem Ergebnisdatensatz nicht unmittelbar entnommen werden, nach wie vielen Zyklen der beabsichtigte Pfad erreicht wurde. Bei speziellen Applikationen im Maschinen- und Anlagenbau ist jedoch auch gerade diese Information von zentraler Bedeutung, da hier bspw. ein Not-Halt innerhalb eines Zyklus gewährleistet sein muss.

Durch die Angabe einer Assertion kann jedoch der Pfad von Interesse vor der Ausführung markiert werden, siehe Zeile 13 in Listing 4.8. Bei der Ausführung von KLEE kann anschließend die Assertion als Identifikationskriterium verwendet werden.

4.5.3.2 Eingrenzen des Variablenbereichs als Transformationsregel

Die Eingrenzung des Wertebereichs der Eingabedaten dient zur Betrachtung von gültigen Werten, die aus Planungsdaten eines Engineering Systems, bspw. EPLAN Engineering Center [EPL13], ein System zur Integration der an der Entwicklung eines mechatronischen Systems beteiligten Disziplinen, extrahiert werden können. Dabei kommt ein Baukastensystem zum Einsatz, mit dem insbesondere die zunehmende Variantenvielfalt im Maschinen- und Anlagenbau abgedeckt werden kann [VHSK07], [FFVH12]. Bei den Werkzeugen der symbolischen Ausführung können die Wertebereiche nicht eingegrenzt bzw. die konkreten Werte nicht spezifiziert werden. Das liegt auch daran, daß es sich nicht um den Anwendungsfall dieser Strukturanalysewerkzeuge handelt. Denn deren primärer Einsatzzweck ist die Bewertung eines Quelltextes bezüglich der Abdeckung der vorhandenen Pfade. Dabei ist nicht die Menge konkreter Eingabedaten von Interesse, sondern ausschließlich die Tatsache, dass es eine gültige Belegung selbiger gibt.

Listing 4.9: Eingrenzen des Wertebereichs am Funktionsbeispiel aus Listing 4.8

```
1  void func( int a, int b )
2  {
3      if ( a >= 0 && a <= 100 &&
4           b >  3 && b <  10 )
5      {
6          static int c = 0;
7          if ( a == 0 && b > 5 )
8              c = 1;
9          else if ( c == 1 )
10         {
11             QX = 0;
12             klee_assert( 0 );
13         }
14     }
15 }
```

Die Eingrenzung der Eingabevariablen wird bei der Verwendung von KLEE über zusätzliche Einschränkungen mit Hilfe einer *if*-Kontrollstruktur vorgenommen. Um

sicherzustellen, dass der zu analysierende Funktionsteil nur dann ausgeführt wird, wenn die Eingabedaten innerhalb des vorgegebenen Bereichs liegen, umhüllt die Kontrollstruktur die Funktion. Ein Beispiel ist in Listing 4.9 dargestellt.

4.5.4 Auswerten der KLEE Berechnungsergebnisse

Das Werkzeug KLEE erzeugt für jeden traversierten Pfad eine .ktest Datei, die die Wertebelegung der Eingabedaten enthält. Dabei kann aus der Datei nicht entnommen werden, welcher Pfad mit den ermittelten Daten durchlaufen wurde. Nach Ausführung der KLEE Werkzeugkette aus Listing 4.3 am Beispiel aus Listing 4.8 erfolgt die Ausgabe in Listing 4.10.

Listing 4.10: KLEE Ausgabe zum Listing 4.8

```
1 KLEE: output directory = "klee-out-46"
2 KLEE: ERROR: /soft/diss/example_chap4.c:14: ASSERTION FAIL: 0
3 KLEE: NOTE: now ignoring this error at this location
4
5 KLEE: done: total instructions = 157
6 KLEE: done: completed paths = 9
7 KLEE: done: generated tests = 8
```

Die KLEE Ausgabe in Zeile 2 verdeutlicht, dass die KLEE Assertion in Zeile 14 erreicht wurde. In Summe wurden 8 Tests generiert, siehe Zeile 7. Die Ergebnisse des Laufs wurden im dem Verzeichnis *klee-out-46* abgelegt, das über *klee-last* erreicht werden kann. In Listing 4.11 ist der Inhalt des Verzeichnisses dargestellt. Darin sind die folgenden Dateitypen enthalten:

- **assembly.ll** LLVM byte code der durch KLEE ausgeführt wurde
- **info** Information zum KLEE Lauf
- **messages.txt** Alle Nachrichten, die KLEE ausgegeben hat
- **run.istats** Globale Statistikinformationen zu jeder Codezeile
- **run.stats** Statistikinformationen von KLEE
- ***.ktest** Enthält die Eingabedaten für jeden gefundenen Pfad
- ***.assert.err** Informationen über den Pfad, in dem KLEE einen Fehler (Assertion) gefunden hat
- ***.pc** Enthält die Constraints zu jedem Pfad im Programm
- **warnings.txt** Warnungen zur KLEE Ausführung

In diesem Beispiel wird die **.assert.err* Datei für die zweite Pfadanalyse erzeugt. Demnach enthält diese entsprechende Datei *test000002.ktest* die Eingabedaten, die zu der Assertion, d.h. zu der zu erreichenden Anweisung führen.

Listing 4.11: Inhalt des klee-last Verzeichnisses zur Ausgabe in Listing 4.10

```
$ ls klee-last
assembly.ll
info
messages.txt
run.istats
run.stats
test000001.ktest
test000002.assert.err
test000002.ktest
test000002.pc
test000003.ktest
test000004.ktest
test000005.ktest
test000006.ktest
test000007.ktest
test000008.ktest
warnings.txt
```

Mit dem letzten Kommando aus der in Listing 4.3 aufgelisteten Werkzeugkette lassen sich die ermittelten Eingabedaten aus der *ktest* Datei ermitteln. In Listing 4.12 ist die Ausgabe aufgeführt. Entsprechend der zeitdiskreten Kodierung durch Nummerierung der jeweiligen Variablen werden für a1, b1, a2, b2 unter dem Schlüsselwort *data* die dazugehörigen Werten ausgegeben. Die Testeingabedaten $T_s(F)$ sind unter 4.9 aufgeführt.

$$T_s(F) = \begin{pmatrix}(a_1, b_1)\\(a_2, b_2)\end{pmatrix} = \begin{pmatrix}(0, 8)\\(0, 0)\end{pmatrix} \tag{4.9}$$

Listing 4.12: Zeitdiskrete Eingabedaten zum Listing 4.8

```
$ bin/ktest-tool.cde --write-ints klee-last/test000002.ktest
ktest file : 'klee-last/test000002.ktest'
args       : ['example_chap4.o']
num objects: 4
object    0: name: 'a1'
object    0: size: 4
object    0: data: 0
object    1: name: 'b1'
object    1: size: 4
object    1: data: 8
object    2: name: 'a2'
object    2: size: 4
object    2: data: 0
object    3: name: 'b2'
object    3: size: 4
object    3: data: 0
```

4.6 Zusammenfassung

In dem vorangegangenen Kapitel wurde ein Konzept zur automatisierten Testdatengenerierung für Fehlersituationen in der Anlage aufgestellt, das die in Kapitel 2 abgeleiteten Anforderungen aufgreift und adressiert. Das Konzept schließt dadurch die im Stand der Technik (Kapitel 3) identifizierten Lücken in der Entwicklung qualitativ hochwertiger Steuerungssoftware als wesentlichen Teil des Gesamtsystems im Maschinen- und Anlagenbau.

Fehler stellen kein absolutes Maß dar. Wie in Kapitel 3.2 beschrieben, agieren Systeme gemäß den physikalischen Gesetzmäßigkeiten und den beabsichtigt und unbeabsichtigt implementierten Verhaltensregeln. Eine Klassifikation des Systemverhaltens in Normal- und Fehlverhalten ist im allgemeinen demnach nicht möglich, sondern bedarf der anwendungsfallbezogenen Interpretation. Das vorgestellte Konzept basiert auf dieser Tatsache und ermöglicht die skalierbare Modellierung und Integration von anlagen- bzw. komponentenspezifischen mechanischen Fehlern während der Systementwicklung. Der zentralen Anforderung der aufwandsarmen und bedarfsgerechten Modellierungstiefe wird über die zweigliedrige Fehlermodellierung in Form einer zustandsdiskreten und mathematisch präzisen Beschreibung Sorge getragen. Durch die Integration des um das Fehlermodell erweiterten technischen Systems in das Gesamtsystemmodell ermöglicht der ganzheitliche Ansatz die modellbasierte Wiederverwendung der Informationen als Grundlage der fehlerbasierten Testdatengenerierung. Abbildung 4.2 stellt die Kopplung des Systemmodells an die Steuerungssoftware dar.

Der Formalismus der Testdatengenerierung vereint die Generierung repräsentativer Testdaten mit hoher Aussagekraft durch den Bezug zum Fehlermodell. Durch die Kopplung des technischen Systemmodells mit der Steuerungssoftware kann über eine definierte Abbildung eindeutig bestimmt werden, an welcher Stelle im ausgehenden Quelltext eine fehlerhafte Komponente angesprochen wird. Eine manuelle Ableitung der Eingabedaten zur gezielten Stimulation des Testobjekts, so dass die intendierte Anweisung erreicht wird, ist eine aufwändige und teilweise nicht lösbare Aufgabe. Mit dem eingeführten Konzept der Testdatengenerierung wird dieser Schritt automatisiert. Die Erreichbarkeit der geforderten Anweisung wird durch mehrfache Stimulation des Testobjekts ermöglicht. Die Bestimmung der Eingabedaten kann einerseits durch eine explizite Berücksichtigung der def-use Abhängigkeit lokaler Variablen des Softwarebausteins erfolgen und andererseits durch den Einsatz eines Werkzeugs zur symbolischen Ausführung.

KAPITEL 5

Realisierung der Testdatengenerierung als vertikaler Demonstrator

Eine zentrale Anforderung der Testdatenermittlung ist deren aufwandsarme Durchführung. Die Automatisierung des im vorangegangenen Kapitel eingeführten Konzepts, bedingt eine softwaretechnische Realisierung. Über den vertikalen Demonstrator werden alle wesentlichen Aspekte der automatisierten Generierung von Eingabedaten für IEC 61131-3 konforme Steuerungssoftware abgebildet. Die Umsetzung stellt einen wesentlichen Bestandteil der Evaluation des Testdatengenerierungsverfahrens dar.

Inhaltsverzeichnis

5.1 Beschreibung des Funktionsumfangs

Zum Einsatz und zur Verifikation des im vorangegangenen Kapitel eingeführten Konzepts zur automatisierten Testdatengenerierung fehlerbehafteter Systeme wird in dem vorliegenden Kapitel eine prototypische Werkzeugunterstützung beschrieben. Diese dient anschließend in Kapitel 6 als Grundlage zur Evaluation der Anforderungserfüllung und generellen Bewertung.

Zur Realisierung des Funktionsumfangs wird ein Softwarewerkzeug entworfen, das die Modellierung des technischen Systems als Teil des Gesamtmodells ermöglicht und in Interaktion mit dem IEC 61131-3 Steuerungsprogramm, die für einen bestimmten mechanischen Fehler notwendigen Testeingabedaten automatisch ableitet. Bei der Umsetzung von Prototypen werden vertikale und horizontale Prototypen unterschieden, siehe Abbildung 5.1. Horizontale Prototypen realisieren dabei eine Schicht vollständig, ohne die zusätzlich dahinter notwendigen Schichten zu implementieren [Bal97]. Dabei wird der horizontale Prototyp meist als Demonstrator bezeichnet und dient für gewöhnlich der nicht technischen Bewertung. Zur Veranschaulichung der wesentlichen Funktionalität eines (Software-)Produkts werden die zentralen Bestandteile entlang aller Schichten prototypisch umgesetzt, was als vertikaler Prototyp („Durchstich“) bezeichnet wird.

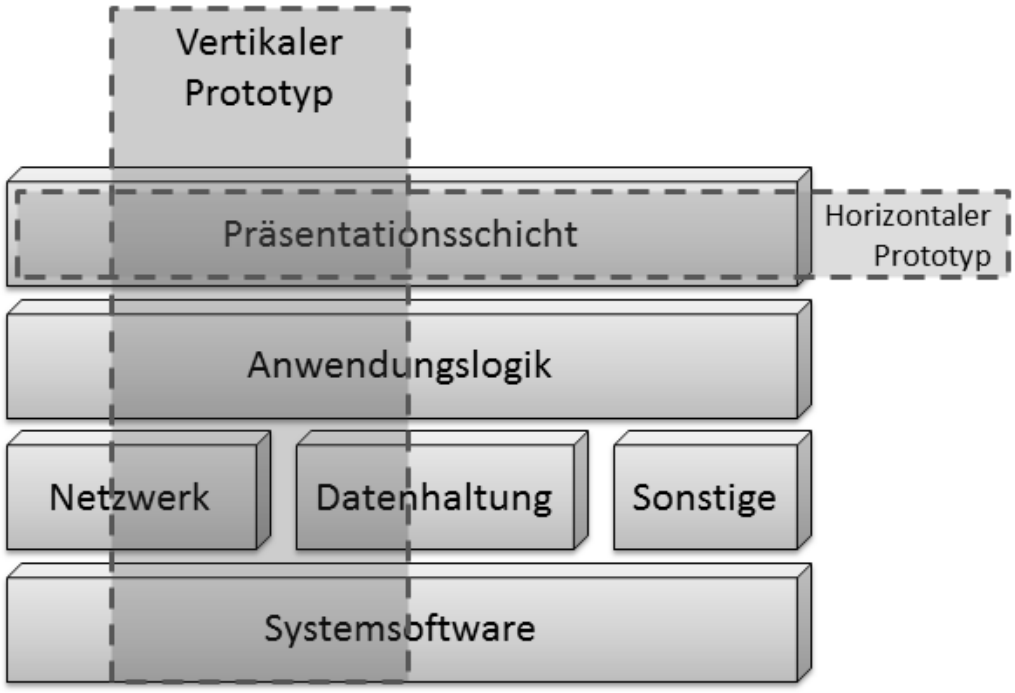


Abb. 5.1: Unterscheidung horizontaler und vertikaler Prototypen (Demonstratoren) [Bal97]

Zur direkten Anwendung des vertikalen Prototyps im Kontext der Entwicklung von IEC 61131-3 Steuerungsapplikationen erfolgt die Realisierung als Funktionserweiterung einer kommerziellen, etablierten IEC 61131-3 Entwicklungsumgebung. Es werden die zur Fehlermodellierung notwendigen Verhaltensdiagramme in den Prototypen

integriert, so dass auf dieser Basis Testdaten über das Generierungsverfahren automatisch abgeleitet werden können. Somit können zur Evaluation des Ansatzes in Kapitel 6 reale, existierende Applikationsbeispiele aufwandsarm analysiert werden, wodurch ein unmittelbarer Interpretationsrückschluss ermöglicht wird. Des Weiteren erfolgt eine Gegenüberstellung der beiden Verfahren zur Generierung der Testeingabedaten aus Abschnitt 4.4 und 4.5 im nächsten Kapitel.

5.2 Softwarearchitektur des Prototyps

In diesem Kapitel werden die Struktur und die einzelnen Komponenten zur Funktionsrealisierung beschrieben.

5.2.1 Logische Struktur und Interaktion der Komponenten

Zur Umsetzung des Konzepts (Abbildung 4.2) als vertikalen Prototypen, müssen die dazu notwendigen Komponenten prototypisch implementiert werden. Abbildung 5.2 zeigt die relevanten Komponenten in einer logischen Struktur.

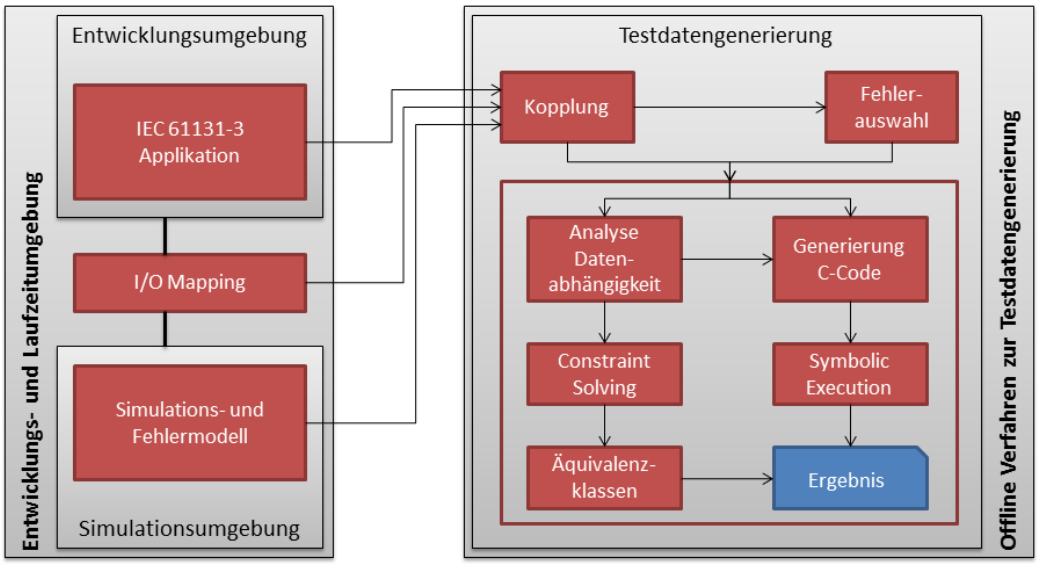


Abb. 5.2: Struktur und Interaktion der Komponenten des Prototypen

Die IEC 61131-3 basierte Steuerungssoftware repräsentiert die Applikationslogik und wird von einer Laufzeitumgebung zur Ausführung gebracht. Durch die Konfiguration der Ein-/Ausgaben (I/O) der Steuerung kommuniziert die Applikation mit der Maschine bzw. Anlage oder einer Simulation [KTVH12]. Das Fehlermodell des technischen Systems enthält die Struktur- und Verhaltensbeschreibungen aller mechanischen Komponenten der realen Anlage und erweitert das Gesamtmodell. Zur Verwendung des Fehlermodells als Basisinformation der Testdatengenerierung für Fehlerfälle muss die Komponente ihre systemrelevanten Informationen zur Verfügung stellen. Durch eine nahtlose Integration in die Entwicklungsumgebung kann eine Verbindung zwischen Ein-/Ausgaben der Steuerung und dem technischen System über ein Software I/O

Mapping realisiert werden. Die einzelnen Komponenten werden in dem nachfolgenden Abschnitt näher spezifiziert.

5.2.2 Beschreibung der Komponenten

Die Komponenten der Entwicklungs- und Laufzeitumgebung sowie der Testdatengenerierung stellen die wesentlichen Bestandteile der Umsetzung des Konzepts dar. Die Steuerungssoftware wird mit Standard-Editoren der IEC 61131-3 Programmiersprachen implementiert und von der Testdatengenerierung analysiert. Das I/O Mapping ist das gängige Verfahren zur Kopplung und Interaktion von Steuerungssoftware und Simulationsmodell (mit Fehlermodell), welches als Basis der Testdatengenerierung dient.

5.2.2.1 Die IEC 61131-3 Steuerungssoftware

Die SPS Steuerungssoftware dient der Funktionsrealisierung der gesamten Maschine bzw. Anlage und besteht aus zahlreichen Softwarebausteinen, um den Geradeauslauf zu realisieren und Ausnahmeroutinen im Fehlerfall auszuführen. Zur Implementierung aller POEs der Steuerungssoftware stehen alle IEC 61131-3 kompatiblen Programmiersprachen zur Verfügung. Die Eingaben des Softwarebausteins können unabhängig von der gewählten Programmiersprache verwendet werden. Die Erbringung der geforderten Funktionalität erfolgt in allen Programmiersprachen semantisch identisch, so dass das Verhalten durch Eingabedaten bestimmt wird.

In SPS Entwicklungssystemen werden IEC 61131-3 Programmfragmente üblicherweise auf eine einheitliche Zwischensprache (ST, AWL o.ä.) abgebildet, siehe Abbildung 5.3. Jede Sprache besitzt jedoch individuelle Umsetzungsregeln, die die Lesbarkeit und das allgemeine Vorgehen der Struktur- und Verhaltensumsetzung vorgeben. Diese Richtlinien gehen bei den Sprachtransformationen verloren, der funktionale Zusammenhang bleibt jedoch erhalten.

Die Testdatengenerierung verwendet Quelltext, der in ST (Strukturierter Text) geschrieben ist. Bei Entwicklungssystemen, die ST als Zwischensprache verwenden, kann das Verfahren auf alle Ausgangsquelltexte angewendet werden. Des Weiteren gibt es keinerlei Vorgaben über die Gestaltung des Steuerungsquelltextes.

5.2.2.2 Das Modell der Komponentenfehler

Das Fehlermodell des technischen Systems ist die Basis für die Generierung aussagekräftiger, reduzierter Datensätze von Testeingabedaten. Das Verhalten von mechanischen Komponenten im Fehlerfall attributiert das Normalverhalten und erweitert somit das Gesamtmodell.

Zur Demonstration des gesamten Testdatengenerierungsansatzes in Form eines vertikalen Prototyps bedarf es der Integration der Komponentenfehlermodelle. Die Modellinformationen dienen in dem präsentierten Gesamtkonzept zur Reduktion der Gesamtheit aller theoretisch ermittelbaren Testdaten und erhöhen somit deren Aussagekraft bei gleichzeitiger Reduktion der Testdatenmenge. Die Beschreibung des Fehlermodells erfolgt in Form von Zustandsdiagrammen [WSVH08], [WRK10]. Damit

wird die zustandsdiskrete Beschreibungsform des Fehlermodells prototypisch realisiert. Das System- und Fehlermodell folgt der festgelegten Namenskonvention, wonach alle Teilmodelle mit dem Präfix $SM\_$ beginnen müssen. Die darin enthaltenen modellierten Fehler besitzen alle das Präfix $err\_$. Bei der prototypischen Implementierung wird dieses Präfix zur Identifikation der Systemmodelle und der modellierten Fehler ausgenutzt.

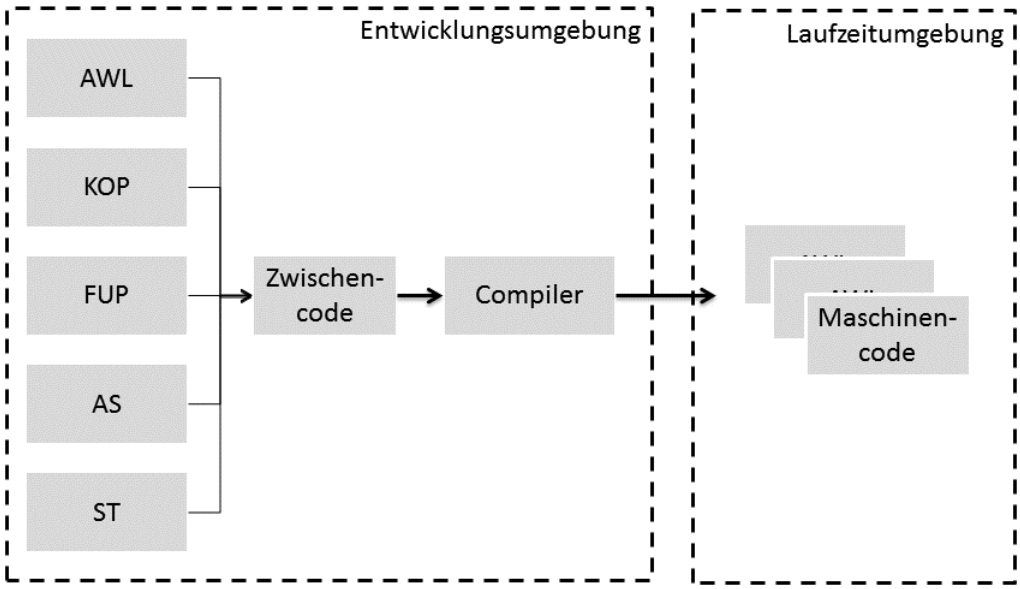


Abb. 5.3: Vereinfachte schematische Darstellung des Übersetzungsvorgangs von IEC 61131-3 Software in Maschinencode

5.2.2.3 Die Kopplung über das I/O Mapping

Zur Verbindung des technischen Systemmodells mit der Steuerungssoftware muss ein Mapping der Ein-/Ausgaben erfolgen. Dabei ist zu unterscheiden, welche technischen Mechanismen miteinander verbunden werden müssen. Bei der Kopplung von Logiken, die in einem Steuerungssystem intern ablaufen, wird für das I/O Mapping ein separater Task und Funktionsbaustein eingesetzt, in dem die Eingänge der Steuerungssoftware den Ausgängen des technischen Systemmodells sowie die Ausgänge der Steuerungssoftware den Eingängen des technischen Systemmodells zugewiesen werden. Dadurch wird eine eineindeutige Zuweisung der korrespondierenden Komponentenvariablen umgesetzt. Die bidirektionale Relation zwischen den beiden Komponenten dient der Testdatengenerierung als Basis.

Die Eingaben der Steuerung werden den Ausgaben des Systemmodells sowie die Ausgaben der Steuerung den Eingaben des Systemmodells zugewiesen, siehe Listing 5.1. Dadurch wird eine bidirektionale Abbildung ermöglicht, so dass von einem zu provozierenden Fehler aus dem Fehlermodell im Systemmodell auf relevante Stellen (Anweisungen) im Steuerungsquelltext rückgeschlossen werden kann. Dies stellt die Grundlage für die zu generierenden Eingabedaten, so dass die durch den Fehler ausgewählte Anweisung des Steuerungsquelltextes erreichbar wird.

Bei der Verbindung eines Steuerungssystems mit einem Simulationssystem erfolgt

die Kopplung meist über vordefinierte Mechanismen des Steuerungs- bzw. Simulationssystems. Eine gängige Form der Anbindung von Simulationssystemen wie bspw. TrySim[1] erfolgt meist durch einen TCP/IP basierten Datenaustausch [Try16].

Listing 5.1: Kopplungsbeispiel von I/O Variablen der Steuerungsapplikation an das Systemmodell mit Hilfe eines separaten Tasks

```
// Erster Zylinder
Modell.Zylinder1.ausfahren := I1;
Modell.Zylinder1.einfahren := I2;
O1 := Modell.Zylinder1.ausgefahren;
O2 := Modell.Zylinder1.eingefahren;

// Zweiter Zylinder
Modell.Zylinder2.ausfahren := I3;
Modell.Zylinder2.einfahren := I4;
O3 := Modell.Zylinder2.ausgefahren;
O4 := Modell.Zylinder2.eingefahren;

// Sensoren
O5 := Modell.MaterialSensor.ws_material;
O6 := Modell.HelligkeitsSensor.ws_material;

// Transportband
Modell.Band.an := I5;
O7 := Modell.Band.Position;
```

5.2.2.4 Automatische Generierung der Testeingabedaten

Die Testdatengenerierung greift für die Berechnung auf die benötigten Basisinformationen (Steuerungssoftware, I/O Mapping, Fehlermodell) zu. Die automatische Generierung der Testeingabedaten erfolgt nach der Auswahl des mechanischen, permanenten Fehlers aus dem Fehlermodell und der dazugehörigen Steuerungssoftware. Unter Berücksichtigung der I/O Kopplung zwischen Fehlermodell und Steuerungssoftware reduziert sich das Problem der Testdatengenerierung als Erreichbarkeit der Anweisung der Steuerungssoftware, die die als defekt markierte Komponente anspricht. Unter Einsatz einer formalen Analyse der Steuerungsapplikation werden die Eingabeparameter (formale Parameterliste und globale Variablen) derart analysiert, so dass deren Wertebelegung die Erreichbarkeit der extrahierten Anweisung ermöglichen.

Der Kontrollflussgraph, die def-use Datenabhängigkeit und die Generierung der Constraint Satisfaction Probleme gemäß Algorithmus 3 sowie die Transformation in ein C Programm erfolgt auf dem abstrakten Syntaxbaum (AST). Für die Auswertung wird

1 TrySim ist ein Simulationswerkzeug, mit dem Maschinen und Anlagen über einfache Geometrien und vordefinierte technische Komponenten zusammengestellt werden können. Mit TrySim können logische Abläufe und Wirkzusammenhänge, insbesondere Fehlerszenarien sehr gut, modelliert und simuliert werden. Es existieren Anbindungen an zahlreiche Steuerungsplattformen.

das Visitor Pattern [GHJ94] verwendet, welches insbesondere zur Traversierung und semantischen Analyse im Compilerbau eingesetzt wird. Das Visitor Pattern kommt genau dann zum Einsatz, wenn gleiche Operationen bei unterschiedlichen Elementen benötigt werden. So treten in einem Syntaxbaum alle syntaktischen Einheiten in Form von Klassen auf, deren Semantik sich kontextabhängig auf die Codegenerierung auswirkt und so individuell interpretiert werden können.

Der abstrakte Syntaxbaum stellt eine Vorstufe des Compilers dar und wird aufgrund der IEC 61131-3 Programmiersprachengrammatik, bestehend aus den Nonterminal- und Terminalsymbolen, der EBNF[1] erstellt. Dabei werden alle im Ausgangsquelltext vorhandenen Ausdrücke, Bedingungen etc. mit Hilfe eines Parsers in die elementaren Symbole der Syntax zerlegt. Alle zur Testdatengenerierung erforderlichen Ableitungen, deren Umsetzung im folgenden Kapitel konkret beschrieben werden, basieren auf den im AST integrierten Informationen.

5.3 Werkzeugimplementierung

Die Implementierung des Werkzeugs erfolgt mit Hilfe des Microsoft .NET Frameworks in der Programmiersprache `C#`. Die entwickelte Software wird dabei nicht mehr nativ in den plattformspezifischen Code übersetzt, sondern in ein unabhängiges Zwischenformat, der Microsoft Intermediate Language (MIL). Die Basisfunktionalität des .NET Frameworks wird neben `C#` für weitere Programmiersprachen angeboten, indem die dazugehörigen Compiler den Quelltext in das Zwischenformat überführen. In Anlehnung an die Plattformunabhängigkeit der Programmiersprache Java wird auch bei .NET kompatibler Software eine Laufzeitumgebung benötigt, die die übersetzten Programmfragmente (Assemblies) zur Ausführung bringt. Diese Aufgabe übernimmt die Common Language Runtime (CLR).

5.3.1 Explizite Betrachtung der Datenabhängigkeit

5.3.1.1 Umsetzung und Integration des Kontrollflussgraphen

Der Kontrollflussgraph wird aus dem gegebenen Softwarebaustein abgeleitet und dient sowohl der Analyse als auch der Datenabhängigkeitsextraktion als Grundlage, siehe Kapitel 5.3.1.1.2. Die Abbildung 5.4 zeigt die Umsetzung und Integration der Datenstruktur des Kontrollflussgraphen (CFG) in Form eines UML Klassendiagramms. Die Basis des CFGs bildet die Klasse *CfgNode* als eine allgemeine Beschreibung eines Knotens. Jeder Knoten eines CFGs kann beliebig viele polymorphe Nachfolgeknoten besitzen. Die jeweilige Ausprägung des allgemeinen Knotens ergibt sich aus den syntaktischen Konstrukten der zugrunde liegenden Eingangssprache (ST) der IEC 61131-3, da diese den Kontrollfluss unmittelbar beeinflussen. Jede individuelle Ausprägung, die in Form der Vererbung realisiert wird, realisiert dabei das angepasste Kontrollflussverhalten und wird für alle in Tabelle 4.2 aufgelisteten Konstrukte (statement, *for* statement, *while* statement, *repeat* statement, *case* statement und *if* statement)

1 Die Extended Backus Naur Form wird zur Beschreibung kontextfreier Grammatik verwendet.

durch die korrespondierenden Klassen umgesetzt. Die Hierarchiebeziehung der Klassen wird durch die reflexive 1:* Relation der Basisklasse *CfgNode* verdeutlicht. Die Methode *Cfg.GenerateCfg()* initiiert die Traversierung des Quelltextes und erzeugt die kontrollflussbezogene Datenstruktur.

Bei der Anwendung des Visitor Patterns wird der vollständige Baum des Ausgangsquelltextes durchlaufen, dessen Struktur analysiert und der dazugehörige Kontrollflussgraph mit den Cfg-Klassen instanziiert. Somit existiert ein vollständiges Abbild des Kontrollflusses nach einem einzigen Auswertungsdurchlauf und dient als Basis für die Erzeugung der Datenabhängigkeit.

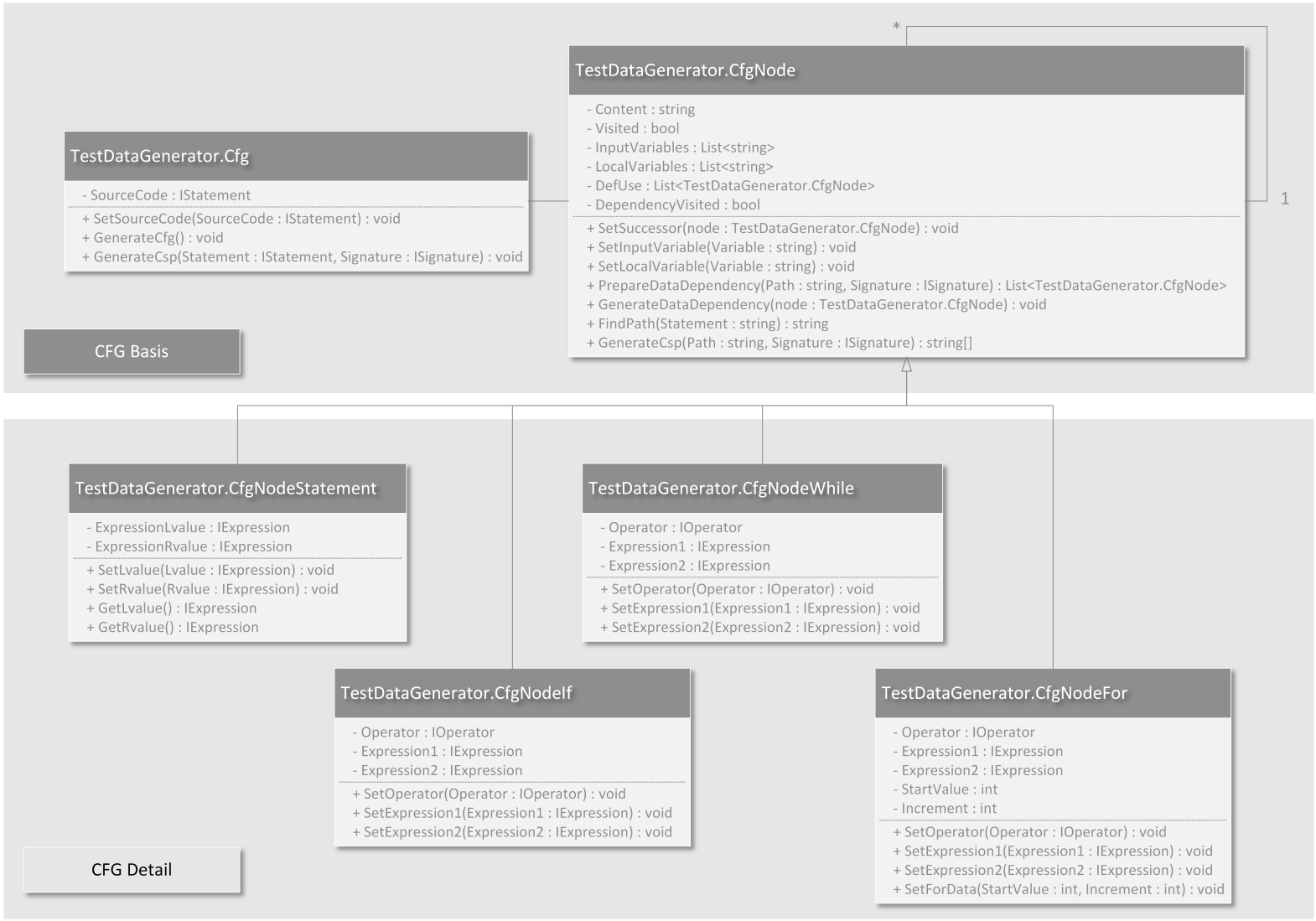


Abb. 5.4: UML Klassendiagramm des Kontrollflussgraphen (Ausschnitt)

5.3.1.2 Umsetzung der Datenabhängigkeitsrelation

Durch die zyklische Ausführungslogik bedingt werden Pfade in IEC 61131-3 basierten Softwarebausteinen erst durch die vorangehende Ausführung (Variablenevaluation) disjunkter Pfade erreicht. Diese unmittelbare Auswirkung auf die Testdatengenerierung bedarf der Berücksichtigung der def-use Datenabhängigkeit zwischen lokalen[1] Variablen.

Die Umsetzung der def-use Abhängigkeit erfolgt auf Knotenebene, indem eine variablenattributierte Kante zwischen den abhängigen Knoten (im Knoten enthaltener Variablen) im CFG eingefügt wird. Die Information über die in den Knoten vorhandenen Variablentypen (lokal, Eingabe, Ausgabe) wird bereits bei der Erstellung des Kontrollflussgraphen durch Abfrage der Signatur in jedem Knoten selektiert und abgelegt. Die Abhängigkeitsrelation wird in der prototypischen Implementierung exakt entlang des Testdatengenerierungspfads bestimmt. Dazu dient die Methode *CFGNode.PrepareDataDependency()*, die alle pfadmanipulierenden Variablen in Bedingungen und Ausdrücken im bottom-up Verfahren von der zu erreichenden Anweisung bis zur Wurzel, d.h. der Startanweisung des Softwarebausteins, analysiert. Anschließend erweitert die Methode *CFGNode.GenerateDataDependency()* den Kontrollflussgraphen um die vorher ermittelte Datenabhängigkeit. Das Beispiel in Abbildung 4.15 verdeutlicht den resultierenden Graphen. Da es bei der Abhängigkeitsbewertung zu mehrstufigen Abhängigkeiten kommen kann, erfolgt diese Auswertungsroutine für jede identifizierte (use) Variable entlang des Pfades rekursiv. Die Suche nach der Definition (def) bzw. Veränderung der use Variablen erfolgt im vollständigen Kontrollflussgraphen. Die Rekursionstiefe liegt damit maximal bei der Anzahl der insgesamt vorhandenen Variablen.

5.3.1.3 Umsetzung der Testdatengenerierung

In Abbildung 5.5 ist der grundlegende Ablauf der Testdatengenerierung in Form eines UML Sequenzdiagramms als Teil der gesamten Kette, siehe Kapitel 5.4, skizziert. Der Ablauf lässt sich im Wesentlichen in drei Teilbereiche einteilen: Konstruktion des Kontrollflussgraphen, Formulierung der Constraint Solving Probleme, Ausführung und Auswertung. Der Kontrollfluss besteht aus einer Objekthierarchie von *CfgNode* Objekten. In zweiten Schritt werden nach der Auswahl des Fehlers und der dazugehörigen Anweisung, die Eingabe- und lokalen Variablen gesetzt, bevor die eigentliche Formulierung der CSPs erfolgt.

Für jedes CSP muss der Pfad für die zu erreichende Anweisung ermittelt werden, um anschließend alle Bedingungen zu extrahieren und in Constraints überführen zu können. Diese werden mit Hilfe einer Tiefensuche identifiziert und als Liste von Kanten zurück gegeben. Sofern Elemente der Liste Datenabhängigkeiten zu anderen Zweigen besitzen, was durch die Methode *PrepareDataDependency()* ermittelt wird, muss der

1 Lokale Variablen können über die Eingabeparameter beeinflusst werden, sie können jedoch auch reine kontrollflussmanipulierende Variablen darstellen. Im zweiten Fall dienen sie meist der Zustandsbeschreibung und können nicht direkt über die Eingabeparameter beeinflusst, sondern müssen durch explizite Ausführung manipulierender Pfade verändert werden.

Kontrollflussgraph um die def-use Relation erweitert werden. Dieser Vorgang wird solange wiederholt, bis es keine Abhängigkeiten mehr innerhalb der Constraints gibt.

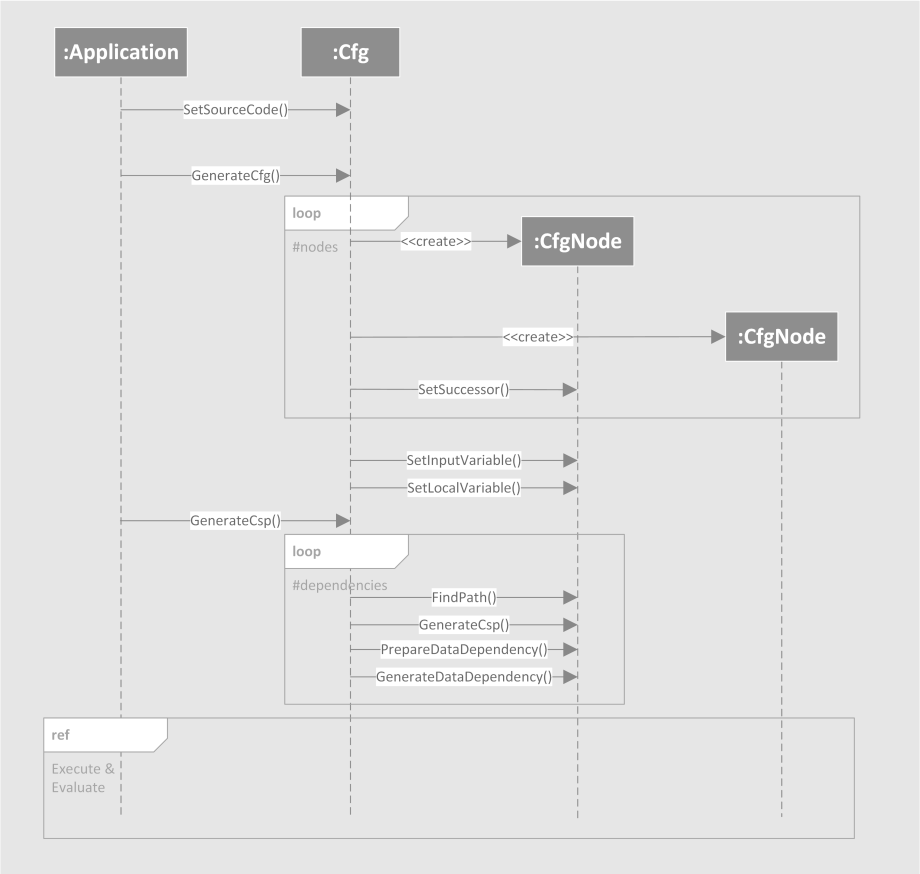


Abb. 5.5: Sequenzdiagramm zum Ablauf der Testdatengenerierung

Im dritten und letzten Schritt werden die aufgestellten CSPs in umgekehrter Reihenfolge ausgeführt, siehe Algorithmus 3, und anschließend die Ergebnisse ausgewertet. Die generierten CSPs folgen dabei den in Kapitel 4.4.2.2 eingeführten Transformationsregeln für den Constraint Solver Minion.

Der Constraint Solver Minion: Für jedes zu lösende CSP wird in den vorangegangen Schritten eine separate Beschreibungsdatei im Minion Format erzeugt. Der dazugehörige Dateiname mit *.minion* Suffix besteht aus dem konstanten Präfix *testdata_*, der um den jeweiligen Index i als Ordnung der CSPs erweitert wird. Die resultierenden CSPs werden in umgekehrter Reihenfolge gelöst. Der Constraint Solver Minion wird aus dem Prototypen heraus mit der Datei des zu lösenden CSPs aufgerufen.

Die Lösungen eines CSPs werden auf der Standardausgabe ausgegeben. Dabei werden die Werte der einzelnen Variablen des CSPs in der unter PRINT der Minion-Datei angegebenen Reihenfolge ausgegeben. Das Listing 5.2 zeigt eine beispielhafte Ausgabe, die für die Auswertung mit einem Parser analysiert werden muß. Die ersten fünf Zeilen sind allgemeine Informationen über die Berechnungszeiten und können für die Auswertung der Lösung vernachlässigt werden. Die Zeilen 10-18 sind im Wesentlichen vernachlässigbar, lediglich Zeile 16 zeigt die gesamte Berechnungsdauer an. In den

letzten beiden Zeilen (hier 19 und 20) wird jeweils angegeben, ob das CSP lösbar ist und wie viele Lösungen gefunden wurden. Die berechneten Lösungen fügen sich unmittelbar an den Header der Ausgabe an und bestehen aus den Variablenwerten (eine je Zeile) und einer abschließenden Lösungsnummerierung, sowie Zeitinformationen zur Berechnungsdauer.

Listing 5.2: Minion Ausgabe der Lösung eines Beispiel CSPs

```
1  Parsing Time: 0.000000
2  Setup Time: 0.000000
3  First Node Time: 0.000000
4  Initial Propagate: 0.000000
5  First node time: 0.000000
6  Sol: 6
7  Sol: 4
8
9  Solution Number: 1
10 Time:0.000000
11 Nodes: 3
12
13 Solve Time: 0.000000
14 Total Time: 0.000000
15 Total System Time: 0.031200
16 Total Wall Time: 0.143000
17 Maximum Memory (kB): 0
18 Total Nodes: 3
19 Problem solvable?: yes
20 Solutions Found: 1
```

Auswertung der Testdatensätze: Beim parameterlosen Aufruf des Minion Constraint Solvers wird die erste gültige Lösung zum CSP, d.h. die kleinste Wertebelegung der Variablen, die alle spezifizierten Constraints erfüllt, ausgegeben. Bei erneutem Berechnungsaufruf werden demnach die exakt gleichen Werte ermittelt. Diese Wertebelegungen sind Repräsentanten der zugehörigen Äquivalenzklassen. Zur Ermittlung aller gültigen Kombinationen und Äquivalenzklassenbestimmung müssen alle Parameterkonstellationen ermittelt werden. Dazu muss Minion mit der Option *-findallsols* ausgeführt werden. Bei der Verwendung dieses Parameters erzeugt Minion alle Lösungen (Variablenwerte) und gibt diese anschließend auf der Standardausgabe nach einer strikten Konvention aus. Analog zur Ausgabe in Listing 5.2 werden, gemäß der Anzahl der gefundenen Lösungen (Zeile 20), die Lösungszeilen (Zeile 6-12) unmittelbar daran anschließend ausgegeben. Das Parsing der Ausgabe ist dabei trivial. Jedoch ist die Formatierung und Darstellung der Ausgabe nicht praxistauglich verwendbar, da die Kombinationsmöglichkeiten der einzelnen Parameterwerte nicht in kompakter Form dargestellt werden. Deshalb wird mit dem Algorithmus aus Kapitel 4.4.5 eine Äquivalenzklassenbildung ausgeführt, so dass die Wertekombinationen im Kreuzprodukt der Parameter auf elementare Untermengen (Äquivalenzklassen) reduziert werden.

Die CSP Ergebnisse werden mit diesem Verfahren für alle *testdata_i.minion* Dateien erfasst, zusammengefasst und in umgekehrter Reihenfolge als zeitdiskrete Testeingabedaten ausgegeben.

5.3.2 Testdatengenerierung durch symbolische Ausführung

Die Umsetzung der Testdatengenerierung mit Hilfe des Werkzeugs zur symbolischen Ausführung (KLEE) ist neben der Datenabhängigkeitsbetrachtung ebenfalls in den Prototypen integriert. Es werden dabei dieselben Mechanismen, wie unter Abschnitt 5.3.1 erläutert, verwendet.

Die beiden zentralen Aufgaben für die symbolische Ausführung stellen die Transformation und die Ausführung bzw. Auswertung durch KLEE dar. In den beiden folgenden Abschnitten werden diese Teilaufgaben näher erläutert.

5.3.2.1 Umsetzung der Quelltext Transformation

Zur Anwendung der symbolischen Ausführung durch KLEE muss der IEC 61131-3 basierte Quelltext von Strukturiertem Text in die Programmiersprache C übersetzt werden. Entsprechend des Konzeptes aus Abschnitt 4.5 muss für alle unterstützten Konstrukte die Transformationsvorschrift realisiert werden. In Abbildung 5.6 ist ein Ausschnitt des UML Klassendiagramms mit den wesentlichen Klassen, Methoden, Attributen und deren Beziehungen zueinander dargestellt. Das Parsen des Ausgangsquelltextes wird über die *KleeTransformation* Klasse umgesetzt. Für jedes zu übersetzendes Element wird ein Objekt der abstrakten Klasse[1] *TransformationElement* mit jeweiligen spezifischen Ausprägung (if, while etc.) erstellt. Für die Transformation des Ausgangsquelltextes wird die abstrakte Methode *SetTransformation* in den darunterliegenden Klassen implementiert. Die Logik der Transformation entspricht den in Abschnitt 4.5.2 aufgestellten Transformationsregeln für Anweisungen/Ausrücke, Kontrollstrukturen, Datentypen und Operatoren.

Ein weiterer wesentlicher Schritt im Rahmen der Transformation stellt die Eingrenzung der Eingabevariablen dar, da diese entgegen der expliziten Betrachtung der Datenabhängigkeit aus Abschnitt 4.4 mit KLEE nicht explizit vorgenommen werden kann. Deshalb müssen die gültigen Bereiche der Eingabevariablen in Form von Bedingungen zu Beginn des transfomierten Quelltextes berücksichtigt werden. Die Restriktionen der jeweiligen Variablen können der *Transformation* Klasse über die Methode *SetInputRestrictions* übergeben werden. In der in dieser Arbeit entstehenden prototypischen Implementierung werden sowohl Intervallgrenzen als auch eine Liste von konkreten Werten als Eingrenzung umgesetzt.

1 Aus einer abstrakten Klasse kann nicht unmittelbar ein Objekt erzeugt werden, sondern nur durch eine konkrete Implementierung der abstrakten Klasse durch eine Kindklasse.

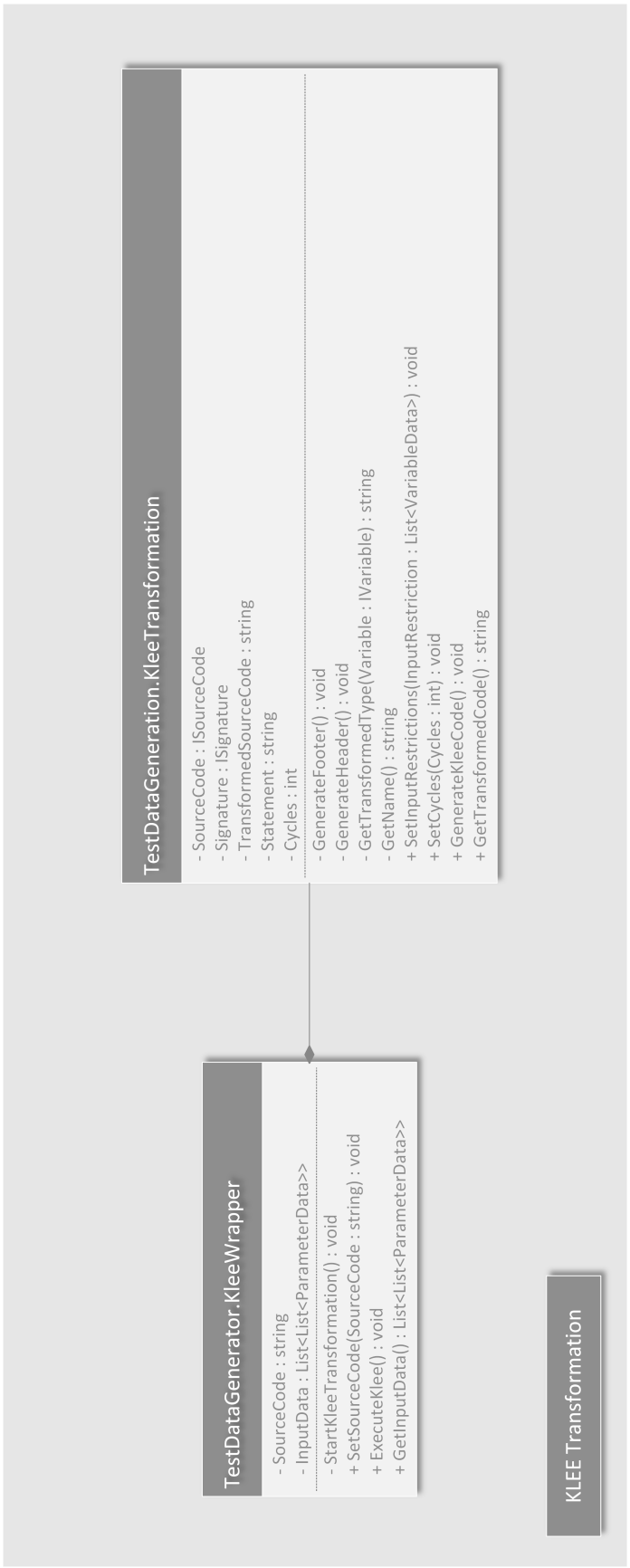


Abb. 5.6: UML Klassendiagramm zur symbolischen Ausführung (Ausschnitt)

5.3.2.2 Umsetzung der Testdatengenerierung durch symbolische Ausführung

Das Werkzeug KLEE ist ein Softwareprojekt, das für Linux basierte Systeme verfügbar ist. Es besitzt mitunter weitere Abhängigkeiten zu Bibliotheken und weiteren Softwarepaketen, die speziell im Linux Umfeld eingesetzt werden. Die prototypische Implementierung der Testdatengenerierung hingegen ist vollständig Windows basiert und kann u.a. aufgrund der starken Bindung zum .NET Framework nicht unter Linux ausgeführt werden. Die Ausführung von KLEE unter Windows ist selbst durch die Verwendung von Cygwin [Cyg14], aufgrund der Abhängigkeiten zu bspw. LLVM (Low Level Virtual machine) und dem Constraint Solver STP nicht unmittelbar möglich. Für die prototypische Umsetzung und zur Evaluierung des Ansatzes genügt es, KLEE konforme Quelltexte zu generieren und diese anschließend manuell auszuführen. Eine automatisierte Kopplung zweier Applikationen auf verteilten Plattformen ist grundsätzlich möglich, ist jedoch für die Bewertung der Leistungsfähigkeit des Ansatzes nicht von zentraler Bedeutung.

Die Ausführung wird durch die Klasse KLEEWrapper vorbereitet, siehe Abbildung 5.6. Die Methode *SetSourceCode()* übergibt dem Wrapper den transformierten C Quelltext zur anschließenden Ausführung von KLEE und Berechnung der Eingabedaten. Eine Automatisierung der Zurückführung der KLEE Berechnungsergebnisse in die prototypische Umsetzung ist möglich, jedoch nicht Bestandteil der Implementierung.

Im Folgenden wird der gesamte Ablauf der Generierungskette exemplarisch beschrieben und durch Screenshots verdeutlicht.

5.4 Ablauf der gesamten Testdatengenerierungskette

Die prototypische Implementierung der Testdatengenerierungskette demonstriert die Ermittlung der zu erreichenden Quelltextanweisung durch die Auswahl eines im Systemmodell hinterlegten Fehlers. Dabei werden keinerlei verhaltensrelevante Aspekte des System- und Fehlermodells benötigt, da zur Testdatengenerierung ausschließlich strukturelle Informationen über die Kopplung des Steuerungsquelltextes mit dem Systemmodell notwendig sind. Die Testdatengenerierung kann auf Steuerungsapplikationen, die in Strukturierten Text geschrieben sind, angewendet werden.

5.4.1 Explizite Betrachtung der Datenabhängigkeit

In Abbildung 5.7 ist der anschließend erscheinende Dialog zur Auswahl des Fehlers bzw. der zu erreichenden Anweisungen abgebildet. Auf der linken Seite des Dialogs ist der aktive Quelltext, in der Mitte die I/O Kopplung und auf der rechten Seite die Fehler im dazugehörigen Systemmodell dargestellt. Der Dialog ermöglicht nun einerseits die Auswahl eines Fehlers, wodurch über die I/O Kopplung der Rückschluss auf ein oder mehrere Stellen im Quelltext möglich ist, als auch die direkte Auswahl einer Anweisung im Quelltext, zu der anschließend die Testeingabedaten generiert werden. Über den Button *Testdaten generieren* wird der Generierungsvorgang mit den ausgewählten Parametern gestartet. Zur Reduktion der Komplexität des Constraint Satisfaction Problems und der gleichzeitigen Optimierung der Performance bei der

Testdatenberechnung können die Wertebereiche der jeweiligen Eingabeparameter manuell verfeinert werden. Da diverse Sensoren und Aktoren für gewöhnlich nicht die vollständige Bandbreite der korrespondierenden Datentypen ausnutzen, ist dieser Schritt auch aus Applikationssicht unterstützend. In der prototypischen Umsetzung wird zwischen einem vollständigen Wertebereich, Teilintervallen oder auch Einzelwerten unterschieden.

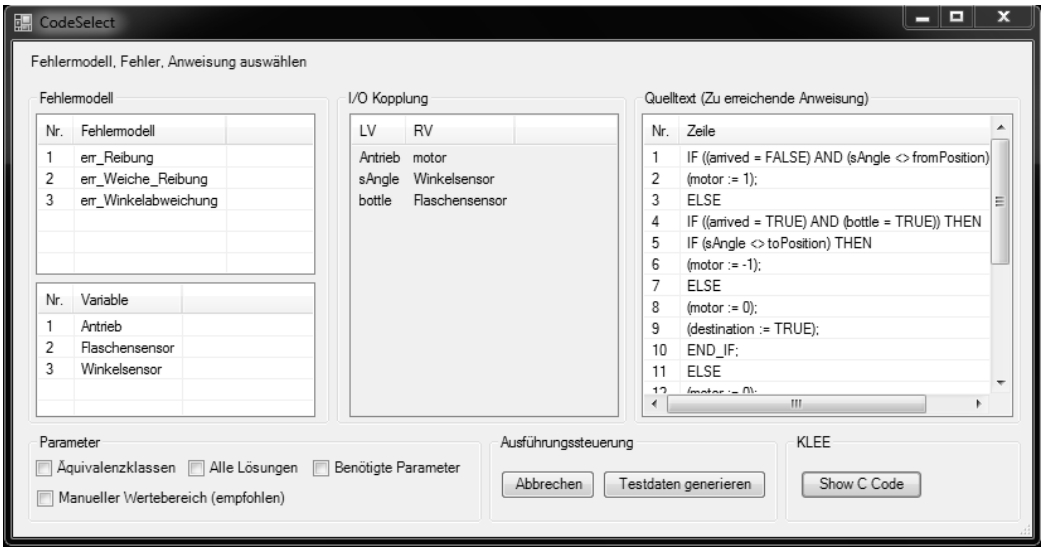


Abb. 5.7: Dialog zur Fehler- bzw. Anweisungsauswahl der Testdatengenerierung

Das Ergebnis der Testdatengenerierung wird nach der Auswertung der Ausgabe des Minion Constraint Solvers, wie in Kapitel 5.8 beschrieben, ausgewertet und anschließend in einem Ergebnisdialog, siehe Abbildung 5.8, dargestellt. Dieser umfasst alle zum Ausgangsquelltext gehörenden Eingabeparameter und die ersten, im Sinne des Wertebereichs niedrigsten, zugehörigen gültigen Werte. Bei einer anschließenden Stimulation des Testobjekts mit den ermittelten Testeingabedaten würde der Ausgangsquelltext so durchlaufen werden, dass die vorher bestimmte Anweisung erreicht und gegen den ausgewählten Fehler getestet werden kann. Bei einer existierenden Datenabhängigkeit zwischen den lokalen Variablen und Eingabeparametern werden die notwendigen zeitdiskretisierten Parameterwerte in der zur Stimulation benötigten Reihenfolge ausgegeben und graphisch über die Nummerierung der ersten Spalte sortiert.

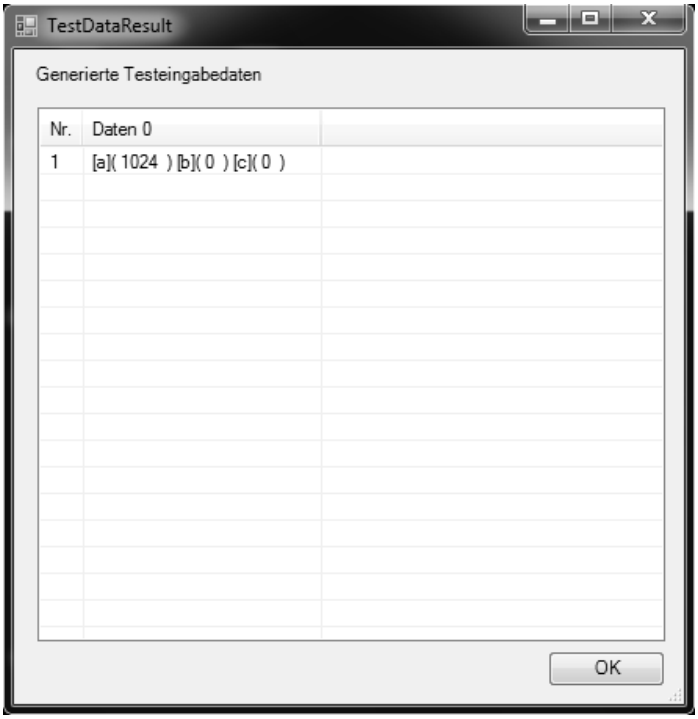


Abb. 5.8: Ergebnisdialog mit den automatisch generierten Testeingabedaten

Bei der Generierung aller Testeingabedaten über den Parameter *-findallsols* des Constraint Solvers Minion werden alle gültigen Lösungen ermittelt und ausgegeben. Über den Generierungsparameter *Äquivalenzklassen bilden* im Dialog in Abbildung 5.7 wird diese Option aktiviert. Durch den erhöhten Berechnungs- und Auswertungsaufwand wird die Implementierung in einem separaten Task[1] ausgeführt, so dass Fortschrittsinformationen in der Oberfläche ausgegeben werden können. Die Darstellung der Ergebnisse bei einzelner Werteberechnung erfolgt analog zur jener der Äquivalenzklassenberechnung, siehe Abbildung 5.9. Für jede Kombination der Äquivalenzklassen wird eine separate Spalte vorgesehen.

5.4.2 Testdatengenerierung durch symbolische Ausführung

Nach der Anwahl der Testdatengenerierung durch symbolische Ausführung erscheint der entsprechende Dialog in Abbildung 5.10. Der Dialog und die Bedienung entspricht dem der expliziten Datenabhängigkeit. Lediglich die Bildung von Äquivalenzklassen ist nicht möglich, da mit dem Werkzeug KLEE nur jeweils eine Lösung erzeugt wird.

1 Bei der Oberflächenprogrammierung ereignisbasierter Systeme ist der Hauptprozess für die Aktualisierung und die Interaktion mit der Oberfläche verantwortlich. Wenn eine umfangreiche Berechnung im selben Kontext ausgeführt wird, friert die Oberfläche ein, und die gesamte Applikation kann abstürzen. Deshalb werden derartig komplexe Berechnungen in einen Hintergrundtask ausgelagert, der Berechnungsergebnisse an die Oberfläche (den Vordergrundtask) kommuniziert.

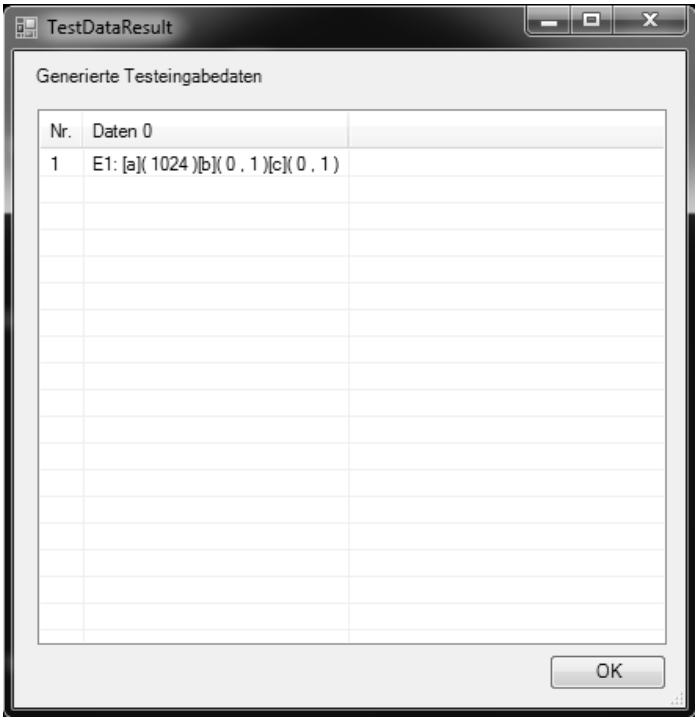


Abb. 5.9: Ergebnisdialog mit den automatisch generierten und in Äquivalenzklassen sortierten Testeingabedaten

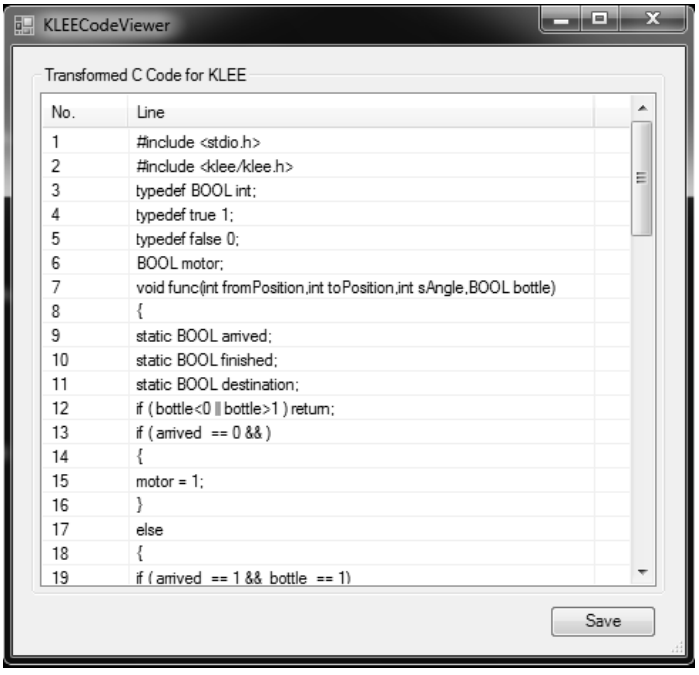


Abb. 5.10: Dialog mit dem transformierten Quelltext zur symbolischen Ausführung

5.5 Zusammenfassung

In dem vorliegenden Kapitel wurde die Realisierung der Testdatengenerierung als vertikaler Prototyp beschrieben. Die Umsetzung erfolgte dabei mit dem Microsoft .NET Framework und der Programmiersprache `C#`. Die logische Struktur der Softwarearchitektur und die Interaktionsrelation der beteiligten Komponenten beschreiben den umgesetzten Funktionsrahmen.

Die Werkzeugimplementierung bildet die wesentlichen Funktionsmerkmale über alle konzeptrelevanten Aspekte ab. Die Modellierung des Gesamtsystems und der darin enthaltenen Fehler erfolgt mit Hilfe der zustandsbasierten Beschreibung. Der gesamte Ablauf der automatisierten Testdatengenerierung wurde vollständig in Eigenentwicklung realisiert, so dass mit dem entstandenen Demonstrator ein Mechanismus zur Anwendung auf IEC 61131-3 Applikationen zur Verfügung steht.

Im nächsten Kapitel erfolgt eine umfassende Bewertung des Konzepts und der Implementierung. Dabei werden insbesondere die erhobenen Anforderungen auf deren Erfüllung überprüft sowie die Erweiterungsmöglichkeiten der Umsetzung diskutiert. Zusätzlich werden die beiden Testdatengenerierungsansätze in einem quantitativen Vergleich bewertet.

KAPITEL 6

Bewertung des Konzepts und des Softwareprototypen

Nachdem in Kapitel 4 das Konzept zur fehlerbasierten Testdatengenerierung aufgestellt und in Kapitel 5 eine prototypische Umsetzung erfolgte, werden beide abschließend auf die Erfüllung der erhobenen Anforderungen evaluiert. Dazu werden zwei Praxisbeispiele eingeführt, die das Potential von Konzept und Werkzeug demonstrieren. Abschließend erfolgt aus den Erkenntnissen eine Verallgemeinerung des Ansatzes.

Inhaltsverzeichnis

In dem vorliegenden Kapitel wird eine praktische Evaluation des aufgestellten Konzepts und der damit einhergehenden prototypischen Implementierung durchgeführt. Für die Bewertung wird der Erfüllungsgrad der Anforderungen *A1*: Testdaten für mechanische, permanente Fehler, *A2*: Automatische Generierung der Testdaten, *A3*: Granularität der Fehlermodellierung, *A4*: Reduktion der Testdaten auf repräsentative Menge, *A5*: Aufwandsarme Modellierung und Generierung aus Kapitel 2.4.2 herangezogen. Des Weiteren erfolgt eine Analyse in Bezug auf die in Kapitel 3.1 abgeleiteten Bewertungskriterien zur spezifischen Bewertung der Systemmodellierung und Testdatengenerierung.

Die Evaluation umfasst sowohl die Bewertung der Anforderungserfüllung als auch eine Einschätzung der Generalisierungsmöglichkeit. Um eine derartige Aussagekraft sicherstellen zu können, werden an repräsentativen Aufgabenstellungen typische Funktionsumfänge der Steuerung von Maschinen und Anlagen betrachtet. Zu jedem Evaluationsbeispiel erfolgt eine Fehlermodellierung mit der Entwicklung des dazugehörigen Steuerungsprogramms, auf deren Basis die Generierung der Testdaten zur anschließenden Auswertung durchgeführt wird.

Damit eine umfassende Bewertung des Funktionsumfangs der Testdatengenerierung möglich ist, werden in den Evaluationsbeispielen die folgenden Szenarien abgebildet:

- **Generieren eines Testeingabedatensatzes** Die Wertebelegungen der Variablen im Testdatensatz werden zur Stimulation des Testobjekts verwendet. Dabei soll im Testdatensatz jede Variable (jeder Eingabeparameter) durch genau einen Wert repräsentiert sein, wenngleich mehrere Wertebelegungen möglich wären.
- **Generieren aller Variablenwerte** Als Erweiterung des Testeingabedatensatzes sollen hierbei alle möglichen Wertebelegungen der Variablen (Eingabeparameter) ermittelt werden. Anschließend sollen diese Werte auch mit Hilfe der Äquivalenzklassenbildung Klassen zugewiesen werden.
- **Erreichbarkeit der Anweisung** Im Rahmen der Testdatengenerierung soll auch ermittelt werden, ob die Anweisung zur Ansteuerung der defekten Komponente überhaupt erreicht werden kann (unerreichbarer Code).
- **Berücksichtigung der Datenabhängigkeit** Je nach technischem Prozess kann eine Anweisung nur aufgrund mehrfacher Stimulation erreicht werden. Dazu muss die Datenabhängigkeit im Steuerungsquelltext berücksichtigt werden.

Tragfähigkeitsbewertungen eines Konzepts unterliegen dem Optimierungsproblem, der Veranschaulichung anhand darstellbarer und überschaubarer Problembeschreibungen und der Übertragbarkeit auf komplexe Sachverhalte. Aus diesem Grund werden in diesem Kapitel Komponenten und Szenarien einer Laboranlage verwendet, deren Aufgaben und Interaktionen repräsentativ in realen Maschinen bzw. Anlagen eingesetzt werden.

Die Experimente zur Evaluation werden mit Hilfe der prototypischen Implementierung durchgeführt. Dabei erfolgt u.a. auch eine Betrachtung der Performance[1] der Testdatengenerierung, Ergebnisauswertung und der Äquivalenzklassenbildung. Dabei werden alle Beispiele auf einem mobilen Rechner mit den folgenden Hard- und Softwaredaten durchgeführt:

- **Hardware** Intel Core i5, Taktfrequenz 1,7 GHz, Arbeitsspeicher 4,0 GB
- **Software** Windows 7 64 Bit Professional, Visual Studio 2013

Die Messungen der Performance werden durch eine Mittelwertbildung aus zehn auf der oben beschriebenen Rechner-Plattform durchgeführten Berechnungen ermittelt.

Die Generierung der Testdaten erfolgt unter Zuhilfenahme des Prototypen. In der prototypischen Implementierung sind sowohl das für das Evaluationsbeispiel relevante zustandsbasierte Fehlermodell als auch der IEC 61131-3 Steuerungsquelltext integriert.

Für die Bewertung der Performance wird jeweils vor Beginn der jeweils betrachteten Aktion die Zeitmessung gestartet und unmittelbar nach deren Beendigung wieder gestoppt. Über die Differenz der Zeitwerte wird somit die zeitliche Komplexität der Berechnung ermittelt. Von besonderer Beobachtung sind dabei die Erstellung der Datenstrukturen (CFG, Abhängigkeitsgraph), die Generierung des bzw. der CSPs, die Lösungsberechnung und die etwaige Äquivalenzklassenbildung. Auf diese Weise kann eine feingranulare Bewertung der atomaren Schritte zur Generierung der Testeingabedaten erfolgen.

6.1 Quantitativer Vergleich der Generierungsverfahren

In diesem Abschnitt werden die beiden in Abschnitt 4.4 und Abschnitt 4.5 beschriebenen Generierungsverfahren quantitativ verglichen. Für diesen Vergleich werden die in Tabelle 6.1 aufgeführten Programme verwendet, die unterschiedlich komplexe Kontrollstrukturen, Abhängigkeiten zwischen Pfaden und je nach Eingabeparameter verschiedene Parameterräume besitzen. Die Beispielprogramme sind im Anhang A aufgelistet.

Die beiden Verfahren wurden genutzt, um die Testdaten für die Beispielprogramme zu berechnen. KLEE generiert Datensätze zur Ermittlung der Code-Abdeckung, d.h. jeweils einen Eingabedatensatz. Die Performance zur Berechnung von einer Lösung kann mit der des Datenabhängigkeitsansatzes verglichen werden, da dieser ebenfalls die Möglichkeit bietet, genau einen Eingabedatensatz zu berechnen. Tabelle 6.2 zeigt die Berechnungsergebnisse für das Datenabhängigkeitsverfahren für eine Lösung, alle Lösungen (alle Variablenwerte) und alle Lösungen mit Äquivalenzklassenbildung. Die Berechnungsdauer für eine Lösung bewegt sich dabei bei allen Beispielen zwischen 130 ms und 318 ms. Das KLEE Verfahren ist durch die entsprechenden Optimierungen für Quelltextanalyse besonders schnell in der Berechnung eines Eingabedatensatzes, siehe

1 Die Realisierung des Demonstrators ist auf Single-Core Berechnungen ausgelegt und wird vollständig auf der CPU durchgeführt.

Tab. 6.1: Beschreibung der Beispielprogramme für den quantitativen Vergleich

Name	Beschreibung	Parameterraum	Lösungen
Example01	Einfache if Struktur	2^{16}	5
Example02	Einfache for Schleife	2^{16}	32768
Example03	Einfache while Schleife	2^{16}	32768
Example04	Verschachtelte if Strukturen	$2^{16} \times 2^{16} \times 2^{16}$	983040
Example05	Verschachtelte for Schleifen	0	-
Example06	Verschachtelte while Schleifen	2^{16}	32767
Example07	Zyklische Abhängigkeit	2×2^{16}	32723
Example08	Doppelte zyklische Abhängigkeit	3×2^{16}	16109
Example09	Hohe Anzahl Funktionsparameter	2^{16}	5
Example10	Bedingte Eingabeparameter	$2^{16} \times 2^{16}$	32768

Tabelle 6.3. Die Berechnung von Eingabedatensätzen erfolgt in weniger als 10 ms, sofern keine zyklische Datenabhängigkeit existiert. Bei Example07 und Example08 steigt die Berechnungsdauer an. Teil des in Abschnitt 4.4 beschriebenen Verfahrens ist die Verwendung der Zyklenanzahl aus der Datenabhängigkeitsanalyse. Würde diese Information im Example08 nicht integriert sein, müsste dieses Beispielprogramm so oft ausgeführt werden (drei mal), bis die hinterlegte Assertion ausgeführt wird. Dadurch würde sich die Berechnungsdauer auf ca. 400 ms summieren. Bei den ermittelten Ergebnissen unterscheiden sich die beiden Verfahren teilweise. Es werden stets gültige Eingabedaten berechnet, jedoch mit dem Datenabhängigkeitsverfahren stets die ersten Werte des Wertebereichs ermittelt, wohingegen KLEE um Werte von Bedingungen variiert. Deutlich wird dies bspw. bei Example07. Eine graphische Darstellung der Ergebnisse des Vergleichs ist in den Abbildungen 6.1 bis 6.3 dargestellt. In Abbildung 6.4 ist der Vergleich der Berechnungsdauern beider Verfahren für eine Lösung dargestellt.

Das Datenabhängigkeitsverfahren ermittelt darüber hinaus auch alle Lösungen, was bei einfachen Beispielprogrammen Constraint Solver bezogen sogar schneller berechnet wird als bei der Ermittlung einer einzelnen Lösung. Bei zunehmendem Lösungsraum steigt die Berechnungsdauer bei den Beispielen an bis auf ca. 60 s. Über das zusätzliche Verfahren der Äquivalenzklassenberechnung wird der Lösungsraum auf Repräsentanten stark reduziert. So werden die insgesamt 32768 Lösungen bei Example10 auf 256 Äquivalenzklassen reduziert.

Um die Grenzen der Berechnung aller Lösungen (Variablenwerte) des Datenabhängigkeitsverfahrens besser einschätzen zu können, wurde ein Programmbeispiel schrittweise um die Anzahl der Eingabeparameter erweitert und der Wertebereich der Eingabeparameter stets so hoch gewählt, dass die Lösung noch berechnet werden kann, siehe Tabelle 6.4. Es zeigt sich, dass die Grenze der Eingabedatenberechnung bei einem Parameterraum bei ca. 2^{16} liegt. In der Tabelle sind bereits die Parameterraumobergrenzen angegeben, für die noch eine Lösung ermittelt werden kann.

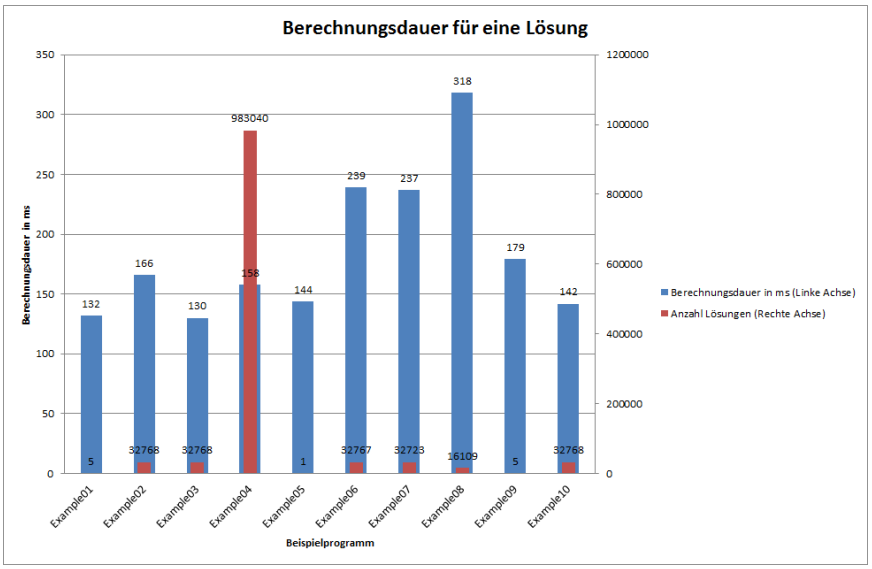


Abb. 6.1: Berechnungsdauer für eine Lösung mit dem Datenabhängigkeitsverfahren

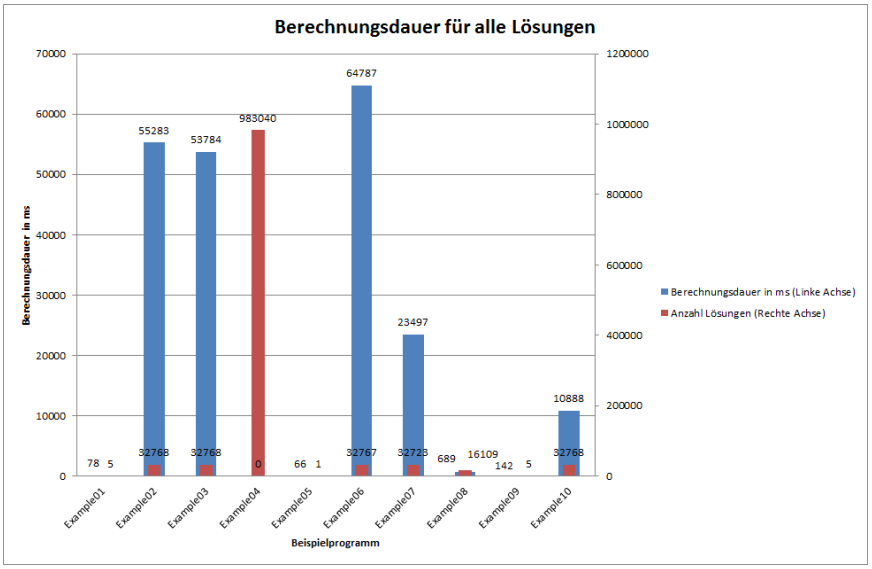


Abb. 6.2: Berechnungsdauer für alle Lösungen mit dem Datenabhängigkeitsverfahren

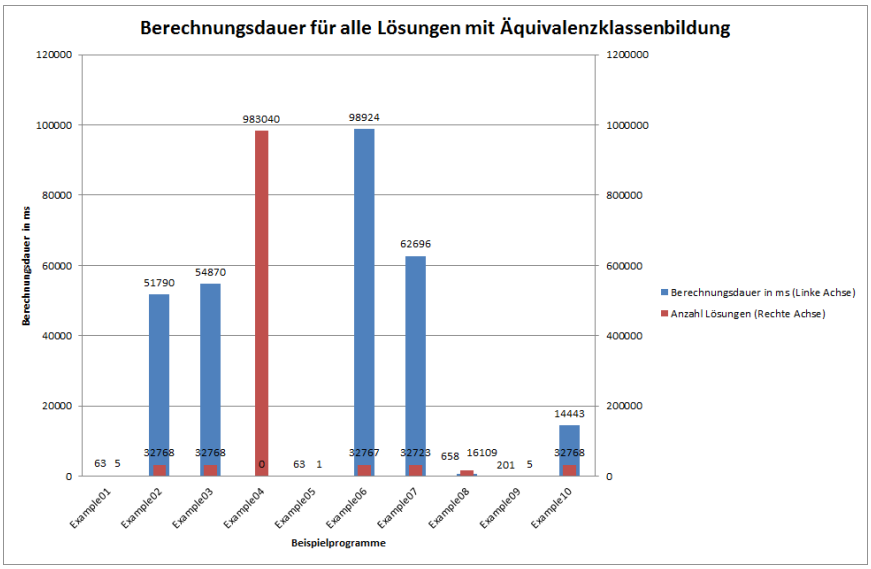


Abb. 6.3: Berechnungsdauer für alle Lösungen mit Äquivalenzklassenbildung mit dem Datenabhängigkeitsverfahren

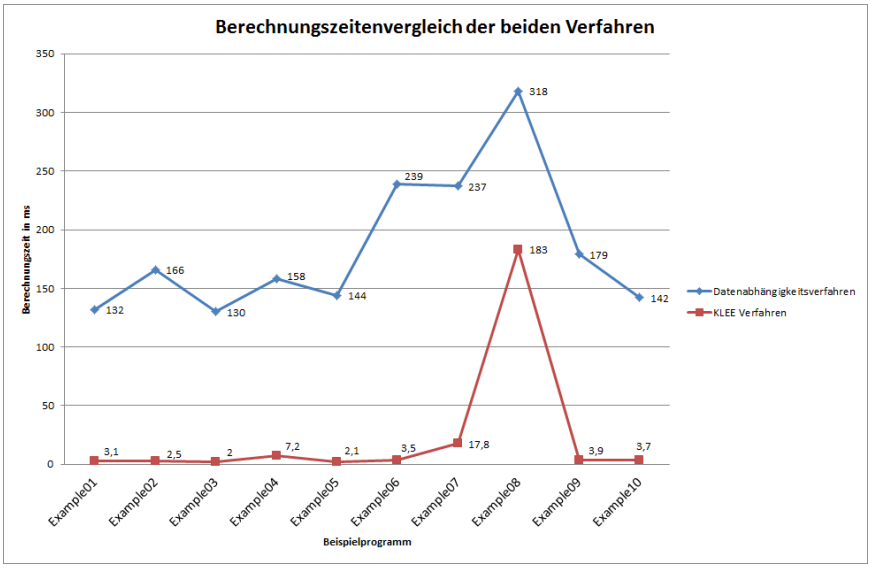


Abb. 6.4: Vergleich der Berechnungsdauer für eine Lösung

Tab. 6.2: Ergebnisse der Testdatengenerierung mit dem Datenabhängigkeitsverfahren

Name	Suchkriterium	Lösung	Zeit [ms]
Example01	Eine Lösung	a(0)	132
	Alle Lösungen	a(0-4)	78
	Äquivalenzklassen	1: a(0-4)	63
Example02	Eine Lösung	a(0)	166
	Alle Lösungen	a(0-32767)	55283
	Äquivalenzklassen	1: a(0-32767)	51790
Example03	Eine Lösung	a(0)	130
	Alle Lösungen	a(0-32767)	53784
	Äquivalenzklassen	1: a(0-32767)	54870
Example04	Eine Lösung	a(0), b(0), c(0)	158
	Alle Lösungen	-	0
	Äquivalenzklassen	-	0
Example05	Eine Lösung	stmt reachable	144
	Alle Lösungen	stmt reachable	66
	Äquivalenzklassen	stmt reachable	63
Example06	Eine Lösung	a(1)	239
	Alle Lösungen	a(1-32767)	64787
	Äquivalenzklassen	1: a(1-32767)	98924
Example07	Eine Lösung	t_0: a(45), t_1: a(30)	237
	Alle Lösungen	t_0: a(45-32767), t_1: a(30)	23497
	Äquivalenzklassen	1: t_0: a(45-32767), t_1: a(30)	62696
Example08	Eine Lösung	t_0: a(0), t_1: a(90), t_2: a(181)	318
	Alle Lösungen	t_0: a(0-180), t_1: a(90), t_2: a(181-269)	689
	Äquivalenzklassen	1: t_0: a(0-180), t_1: a(90), t_2: a(181-269)	658
Example09	Eine Lösung	a(0)	179
	Alle Lösungen	a(0-4)	142
	Äquivalenzklassen	1: a(0-4)	201
Example10	Eine Lösung	a(1), b(0)	142
	Alle Lösungen	[a(1),b(0)],[a(2),b(0)],[a(2),b(1)]...	10888
	Äquivalenzklassen	1: [a(1),b(0)], 2: [a(2),b(0,1)], ... 256:	14443

Tab. 6.3: Ergebnisse der Testdatengenerierung mit KLEE

Name	Abdeckung (I, B)	Lösungen	Zeit [ms]
Example01	100 %, 100 %	a(0)	3,1
Example02	80 %, 50 %	a(0)	2,5
Example03	80 %, 50 %	a(0)	2,0
Example04	100 %, 100 %	a(0), b(0), c(0)	7,2
Example05	65,85 %, 50 %	0 (erreichbar)	2,1
Example06	100 %, 100 %	a(2)	3,5
Example07	100 %, 100 %	t_0: a(46), t_1: a(30)	17,8
Example08	100 %, 100 %	t_0: a(90), t_1: a(90), t_2: a(265)	183,0
Example09	100 %, 100 %	a1(0), a2(0), ... , a16(0)	3,9
Example10	100 %, 100 %	a(1), b(0)	3,7

Tab. 6.4: Berechnung aller Lösungen mit zunehmender Parameteranzahl im Datenabhängigkeitsverfahren

Name	Parameterraum	Zeit [ms]
Example11	2^{16}	14000
	2^{10}	103
	2^{8}	74
	2^{6}	83
	2^{4}	71
Example12	$2^{8} \times 2^{8}$	2077
	$2^{6} \times 2^{6}$	245
	$2^{4} \times 2^{4}$	195
Example13	$2^{4} \times 2^{4} \times 2^{4}$	264
Example14	$2^{4} \times 2^{4} \times 2^{4} \times 2^{4}$	2666

6.2 Evaluation am Beispiel eines Elektromotors

In diesem Evaluationsbeispiel wird die Funktionalität eines Transportbandes betrachtet. Bei realen Produktionsanlagen sind Transportfunktionen elementarer Bestandteil der Funktionsrealisierung.

6.2.1 Kurzbeschreibung des Praxisbeispiels

In Anlagen werden üblicherweise Güter (Flaschen, Päckchen o.ä.) über diverse Transportbänder bewegt. Typische mechanische Fehler bei Transportbändern sind Verklemmungen und erhöhter Schlupf, die Veränderungen im Prozessablauf reaktiver Systeme nach sich ziehen können. Dabei kann es von einfachen Schlupfsituationen beim Anfahren bis hin zum vollständigen Ausfall der Transportfunktionalität kommen. Diese Ursachen können zu einer Überlastung des Motors und so zu einem vollständigen Ausfall des Teilsystems führen. Im nächsten Unterkapitel wird der Fehler eines Motorausfalls aufgrund einer Überlastung, die bspw. durch eine Verklemmung hervorgerufen wird, modelliert und stellt die Grundlage der Testdatengenerierung.

6.2.2 Fehlermodell und Steuerungsquelltext

Die Qualitätsmerkmale von Elektromotoren, wie bspw. der Sicherstellung einer optimalen thermischen Auslastung, stellen wesentliche Kriterien dar, die durch die Kommutierung beeinflusst werden. Während des Betriebs eines Elektromotors entstehen Wechselwirkungen im gesamten System, deren Intensität durch das Drehmoment und die Drehzahl kontrolliert werden können. Bei einer Verwendung des Motors im erlaubten Dauerbetriebsbereich, siehe Abbildung 6.5, d.h. unterhalb der kritischen Obergrenzen, erfolgt ein kalkulierter Verschleiß im Kommutierungssystem. Eine Belastung außerhalb des Grenzbereichs ist durchaus üblich und führt lediglich bei dauerhafter Beanspruchung zu hohen Strömen, die zu einem Ausfall des Motors führen. Bei der mechanischen Kommutierung bürstenbehafteter Motoren entsteht der notwendige Strom durch Schleifkontakte, deren mechanischer Verschleiß von der Stromhöhe abhängt. Diese Art der Überbelastung dient als Grundlage der Fehlermodellierung.

6.2.2.1 Das zustandsbasierte und parametrische Fehlermodell

Der Elektromotor kann bei einer überdimensionierten kurzfristigen Auslastung, den höheren Verschleiß für einen kurzen Zeitraum kompensieren. Bei der Dauerbelastung kann es jedoch zum Ausfall kommen. In Abbildung 6.6 ist der wesentliche Zusammenhang der Drehmomenterzeugung des in einem Transportband angebrachten Elektromotors im SysML Parameterdiagramm beschrieben. Die präzise Modellierung der Regelung ist im Sinne der exemplarischen Fehlermodellierung (Ausfall Motor) nicht notwendig und auch aus Gründen der Darstellbarkeit in dem Beispiel nicht weiter ausgeführt.

Das Drehmoment eines drehenden Körpers errechnet sich aus dem Produkt der Trägheit und der Winkelbeschleunigung $M = J \cdot \alpha$. In Analogie zur Regelung kann auch die Winkelbeschleunigung als Änderung der Winkelgeschwindigkeit mit $\alpha = \frac{\mathrm{d}\omega}{\mathrm{d}t}$

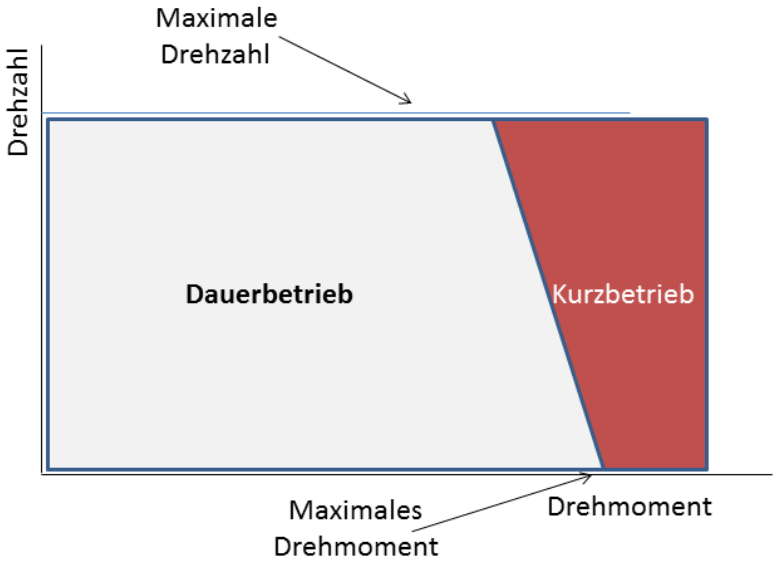


Abb. 6.5: Schematische Darstellung der Belastungsgrenzen eines Elektromotors

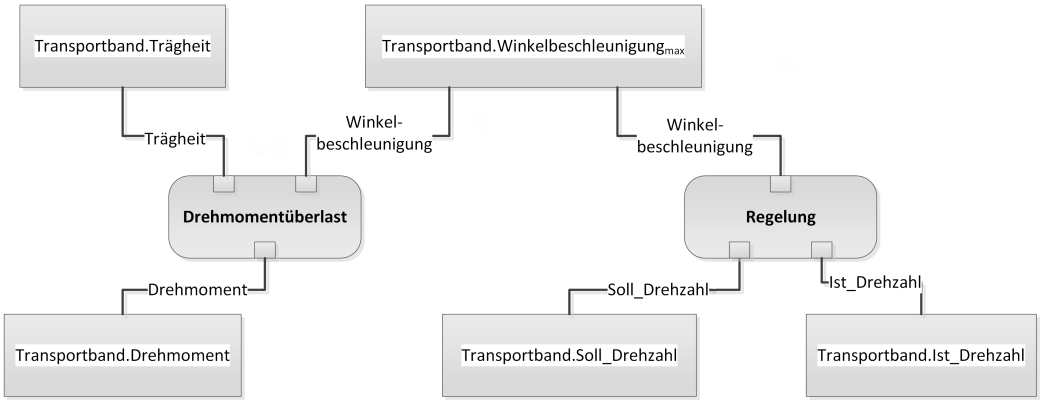


Abb. 6.6: Parametrisches Fehlermodell des Transportbandes (E-Motor)

formuliert werden. Abbildung 6.7 zeigt die Constraints zum parametrischen Fehlermodell, die diese Zusammenhänge darstellen und das Drehmoment durch die maximale Winkelbeschleunigung erhöht.

«constraint» **DrehmomentÜberlast**
Constraints { Drehmoment = Trägheit * Winkelbeschleunigung$_{max}$ }
parameters Drehmoment : Nm Trägheit : kg m² Winkelbeschleunigung : rad/s²

«constraint» **Regelung**
Constraints { Winkelbeschleunigung = Regelung(Soll_Drehzahl, Ist_Drehzahl) }
parameters Winkelbeschleunigung : rad/s² Soll_Drehzahl : rad/min Ist_Drehzahl : rad/min

Abb. 6.7: Constraints zur parametrischen Fehlermodellierung zu Abbildung 6.6

Die zustandsbasierte Verhaltensspezifikation für den Normalbetrieb, inklusive des Fehlermodells, ist in Abbildung 6.8 dargestellt. Der Normalbetrieb lässt sich dabei in drei wesentlichen Phasen abstrahieren, nämlich einen Zustand bei Stillstand, einer

konstanten Bewegung und der Phasen im Übergang zwischen diesen beiden Zuständen. Die Fehlermodellierung ist so umgesetzt, dass der Fehler in jedem Zustand parallel auftreten kann und so zu einem veränderten Verhalten führt. Durch zusätzliche Maßnahmen der Synchronisation zwischen Normalbetrieb und Fehlermodell (einfache Synchronisationsvariablen) kann der Drehmomentüberlast-Zustand so koordiniert werden, dass er ausschließlich im Beschleunigungsfall greift. Das hat allerdings zur Folge, dass Eingriffe in die Verhaltensbeschreibung des Normalfalls getätigt werden und so der Orthogonalitätsfaktor (Unabhängigkeit) der Fehlermodellierung gebrochen wird. Dadurch würden zusätzlicher Wartungsaufwand generiert und u.a. Verantwortlichkeiten gemischt werden.

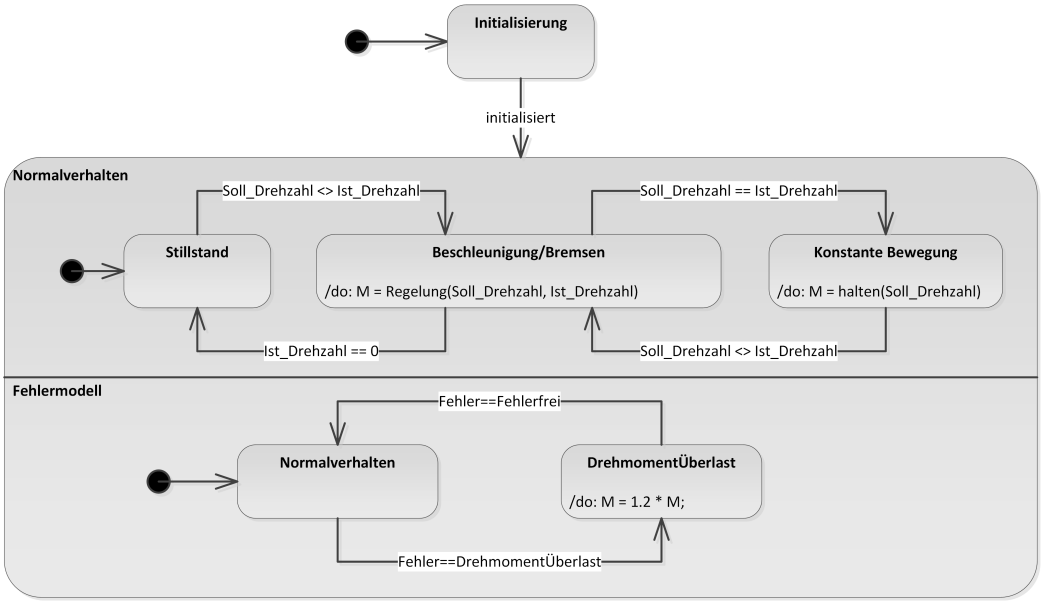


Abb. 6.8: Zustandsbasierte Beschreibung und Integration des Fehlers

6.2.2.2 Der zu analysierende Steuerungsquelltext

Der diesem Beispiel zugrunde gelegte Steuerungsquelltext hat insgesamt einen Umfang von 27 LOC[1]. Das Transportband ist zwischen Weichen positioniert und kann eintreffende Flaschen, sobald der Sensor deren Ankunft registriert, in beide Richtungen transportieren. Der Steuerungsquelltext enthält keine für die Testdatengenerierung relevante Datenabhängigkeit, so dass die gewählten Anweisungen jeweils unmittelbar aus einer einmaligen Berechnung ermittelt werden können. Die für die Bearbeitung dieses Beispiels relevanten Variablen lauten:

- **Eingabevariablen** destination (INT), bDBarCode (INT)
- **I/O** LS_Eingang (BOOL), antrieb (INT), cBBarCode (INT)
- **Lokale Variablen** convey (BOOL)

1 LOC bedeutet Lines Of Code und stellt eine quantitative Metrik dar.

Listing 6.1: Steuerungsquelltext eines Transportbandes für bilateral eintreffende Transportgüter

```
01: IF NOT convey THEN
...
10:     ELSIF destination = 0 THEN
11:             antrieb := -1;
12:             IF LS_Eingang AND cBBarCode = bDBarCode  THEN
13:                     antrieb := 0;
14:                     convey := TRUE;
15:             ELSE
...
27: END_IF
```

6.2.3 Generierung der Testdaten

Bei der Auswahl des Drehmomentüberlast-Fehlers ergibt sich aufgrund mehrfacher Ansteuerung innerhalb des Steuerungsquelltextes über die I/O Kopplung eine Mehrdeutigkeit, die nicht automatisch aufgelöst werden kann. Deshalb wird der Benutzer des Prototypen gefragt, für welche Stelle (Anweisung) Testeingabedaten generiert werden sollen. Für die Testdatengenerierung in Kapitel 6.2.3.1 und 6.2.3.2 wird Zeile 13 des Listings 6.1 ausgewählt. In dem vorliegenden Beispiel wird ein einzelner Testeingabedatensatz generiert sowie die Bildung von Äquivalenzklassen bei der Generierung aller möglichen Lösungen untersucht.

6.2.3.1 Generieren eines Testeingabedatensatzes

Nach Auswahl der zu erreichenden Anweisung wird die Testdatengenerierung mit eingeschränkten Wertebereichen der Eingabeparameter ausgeführt. In diesem Evaluationsbeispiel existieren einige Eingabeparameter vom Datentyp Integer, dessen Wertebereich deutlich größer als für die Applikation notwendig ist. Da sich der Parameterraum aus dem Kreuzprodukt der Wertebereiche aller Eingabevariablen zusammensetzt, ergibt sich ein exponentielles Problem, welches bereits bei zwei bis drei Variablen dieser Art zu einer hohen zeitlichen Berechnungskomplexität führt. Zur Reduktion dieser Komplexität werden die Wertebereiche der Variablen *cBBarCode* und *bDBarCode* auf *[1000,1199]* eingegrenzt.

Die Berechnung der Testeingabedaten liefert die folgenden Wertepaare: *[destination](0), [cBBarCode](1000), [LS_Eingang](1), [LS_Ausgang](0), [bDBarCode](1000)* als einzigen Datensatz. Bei Betrachtung des Ausgangsquelltextes lässt sich schnell vor Augen führen, dass mit der errechneten Eingabewertebelegung alle Bedingungen entlang des Pfades bis hin zur Anweisung in Zeile 13 wahr werden und so die potentielle fehlerhafte Komponente angesprochen werden kann.

In Tabelle 6.5 sind die Messergebnisse der Performance für die jeweiligen Prozessschritte notiert. In diesem Evaluationsbeispiel erfolgt die Erstellung des Kontrollflussgraphen in ca. 10 ms. Die anschließende Formulierung bzw. Ableitung des Constraint Satisfaction Problems erfolgt als Analyseergebnis des Kontrollflussgraphen in ähnlichem zeitlichen Umfang. Mit ca. 230 ms erfolgt die Berechnung des CSPs

Tab. 6.5: Performanceergebnisse des Evaluationsbeispiels für die einzelnen Prozessschritte der Testdatengenerierung (ein Datensatz)

Prozessschritt	**Berechnungsdauer**	
	Datenabh.	KLEE
CFG erstellen	10,4 ms	
CSP formulieren	13,6 ms	
Lsg berechnen	232,5 ms	11,5 ms
Ausgabe parsen	0,0 ms	0,0 ms
Ergebnis verarbeiten	0,8 ms	0,7 ms
Summe	**257,3** ms	**36,2** ms

bzw. in 11,5 ms die Lösungsberechnung. Das ist der eigentliche Generierungsschritt. Das Parsen der ausgegebenen Ergebnisse konnte bei der Performancemessung nicht erfasst werden, da dieser Prozessschritt in weniger als 100 µs abgeschlossen ist. Auf die Ergebnisaufbereitung entfallen eine Rechenzeit von weniger als 1 ms. In Summe erfolgt die gesamte Berechnung in ca. 260 ms bzw. ca. 40 ms.

6.2.3.2 Generieren aller gültigen Testeingabedaten

Analog zum vorangegangenen Fall wird auch bei der vollständigen Testeingabedatengenerierung der Wertebereich auf 200 gültige Werte der Bar Codes eingegrenzt. Im vorangegangenen Fall wurde ein gültiger Testeingabedatensatz erzeugt. Die Generierung aller gültigen Eingabewerte stellt einen komplexeren Berechnungsaufwand dar, liefert jedoch die vollständigen Werte der Eingabedaten zur anschließenden Stimulation. Durch die im Konzept integrierte automatische Äquivalenzklassenbildung werden die ermittelten Intervalle über alle Eingabeparameter (CSP-Variablen) miteinander kombiniert. Je nach Steuerungsquelltext können sich dabei ein oder auch mehrere Äquivalenzklassen ergeben.

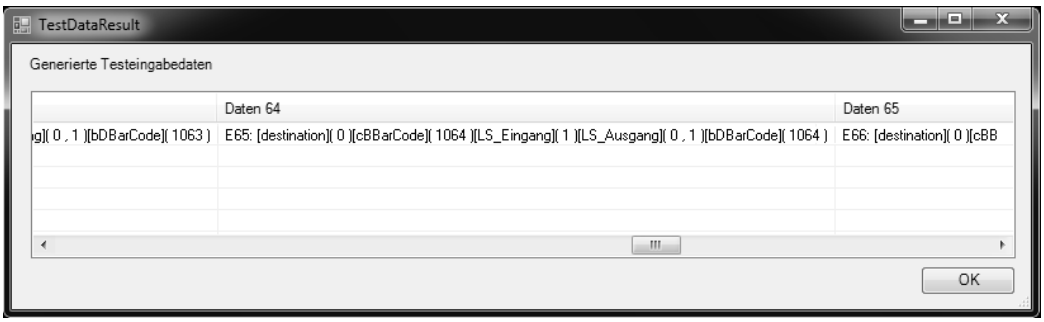


Abb. 6.9: Ergebnisdialog mit allen errechneten Testeingabedaten

Die Berechnung aller Lösungen erfolgt analog zur Generierung aus dem vorangegangenen Kapitel, jedoch mit aktivierter Äquivalenzklassenbildung. Die resultierenden Äquivalenzklassen zu dieser Berechnung sind im Ergebnisdialog in Abbildung 6.9 aufgelistet. Das Ergebnis umfasst insgesamt 201 Äquivalenzklassengruppen, da über die Bedingung $cBBarCode = bDBarCode$ alle identischen Wertebelegungen eine Lösung

repräsentieren, die bspw. unabhängig von der Belegung der Variablen *LS_Ausgang* sind. Die Belegungen der restlichen Variablen bleiben konstant.

Tab. 6.6: Performanceergebnisse des Evaluationsbeispiels für die einzelnen Prozessschritte der Testdatengenerierung (Äquivalenzklassenbildung)

Prozessschritt	**Berechnungsdauer**
CFG erstellen	21,1 ms
CSP formulieren	38,9 ms
CSP berechnen	653,7 ms
Ausgabe parsen	4,1 ms
Ergebnis verarbeiten	241,3 ms
Summe	**959,1** ms

Die Berechnungsdauer aller zur vollständigen Testdatengenerierung erforderlichen Prozessschritte ist in Tabelle 6.6 aufgelistet. Bei der Berechnung der gesamten Eingabedaten entfallen $\frac{2}{3}$ der gesamten Dauer auf die Berechnung des Constraint Satisfaction Problems und der Großteil des verbleibenden Drittels auf die Verarbeitung des Ergebnisses, d.h. der Äquivalenzklassenbildung. Der Constraint Solver liefert weitere detailliertere Zeitinformationen über die CSP Berechnung, woraus ersichtlich wird, dass die eigentliche Berechnung ca. 20 % schneller abläuft als die Ausgabe der Ergebnisse. Die vollständige Kette der Testdatengenerierung wird in weniger als 1 s abgeschlossen und erfüllt somit die gestellte zeitliche Anforderung.

6.2.4 Zusammenfassung

Für das in diesem Kapitel gewählte Evaluationsbeispiel wurden gemäß Konzept ein Fehlermodell für ein überhöhtes Drehmoment eines Elektromotors erstellt. Darauf aufbauend wurden in zwei Verfahren Testeingabedaten automatisch generiert. Die Auswahl des Fehlers führte zu einer Mehrdeutigkeit im Steuerungsquelltext, so dass diese durch eine zusätzliche Selektion konkretisiert werden musste. Die Testdaten konnten in beiden Generierungsläufen in weniger als 1 s erzeugt werden, wohingegen bei der Berechnung aller Lösungen (Variablenwerte) trotz Einschränkung der Wertebereiche bereits ein deutlicher Zuwachs der zeitlichen Komplexität sichtbar wurde. Die Ursache dafür liegt in der Gleichheitsbedingung der beiden Variablen *cBBarCode* und *bDBarCode*.

Zur Kompensation dieses Effekts ist ein mögliches Optimierungspotential die Berücksichtigung der Gleichheitsbeziehung vor oder nach der eigentlichen CSP Berechnung. Bei einer Elimination der Bedingung vor der Berechnung könnte diese Erkenntnis nachträglich der Lösung hinzugefügt werden, was im trivialen Fall eine offensichtliche Möglichkeit darstellt. Da eine Gleichheitsbeziehung jedoch auch Auswirkungen auf alle restlichen Bedingungen haben kann oder auch Datenabhängigkeiten zwischen den beteiligten Variablen und weiteren Zweigen des gesamten Ausgangsquelltextes existieren können, ist diese Variante nicht zielführend. Ein vielversprechenderer Ansatz ist, eine nachgelagerte Analyse der Äquivalenzklassen durchzuführen und derartige zusätzliche Beziehungen automatisch zu identifizieren.

Im Folgenden wird eine Betrachtung des Evaluationsbeispiels im Hinblick auf die gestellten Anforderungen aus Kapitel 2.4.2 vorgenommen.

- **A1** In dem Evaluationsbeispiel konnten permanente, mechanische Fehler des Transportbandes als Konsequenz eines überbelasteten Elektromotors sowohl in der parametrischen als auch in der zustandsbasierten Variante dargestellt werden.
- **A2** Die Generierung der Testdaten erfolgte nach Auflösung der Mehrdeutigkeit automatisch. Dabei lieferte die Generierung eines einzelnen Datensatzes mit ca. 10 ms pro Zeile Quelltext die gültigen Testeingabedaten.
- **A3** Die Granularitätsstufen der Fehlermodellierung ermöglichten in diesem Beispiel eine anforderungsbezogene Fehlerrepräsentation. In der parametrischen Variante konnte der mathematische Zusammenhang des Drehmoments mit der Winkelbeschleunigung, gemäß den physikalischen Gesetzmäßigkeiten spezifiziert werden. Die zustandsbasierte Modellierungsform bildet das Verhalten auf einem Abstraktionsniveau nach.
- **A4** Durch die Äquvalenzklassenbildung auf dem vollständigen Lösungsdatensatz erfolgt eine Clusterbildung über alle Variablenwerte. Alle Kombinationen aus den jeweiligen variablenbezogenen Werten ergeben einen gültigen Eingabedatensatz. Die Mächtigkeit der Äquivalenzklassen lassen Rückschlüsse auf die Bedeutung der darin enthaltenen konkreten Variablenwerte zu, so dass ein jeweiliger Repräsentant die Relevanz des Datensatzes erhöht.
- **A5** Die Fehlermodellierung ist konzeptionell darauf ausgelegt, dass sie orthogonal zur Verhaltensbeschreibung im Normalbetrieb dem Gesamtmodell hinzugefügt werden kann. Durch die bedarfsbezogene Berücksichtigung der als relevant betrachteten Fehler ist der Aufwand gemessen am Gesamtmodell marginal. Der Prozess der Testdatengenerierung erfolgt selbst im komplexeren Fall der vollständigen Datensatzerzeugung in weniger als 1 s.

6.3 Evaluation am Beispiel einer Verpackungsanlage

Das folgende Evaluationsbeispiel besteht aus einer Transportstrecke, über die Päckchen transportiert und bearbeitet werden. Derartige Abläufe und Prozessbearbeitungsschritte stellen ein typisches Muster in Verpackungsanlagen dar. Dieses Beispiel wurde zusammen mit dem Verfahren zur Generierung der Simulationsparameter in [KS16] veröffentlicht.

6.3.1 Kurzbeschreibung des Praxisbeispiels

Das in diesem Abschnitt gewählte Applikationsbeispiel ist vollständig in dem Simulationswerkzeug TrySim [Try16] realisiert. Bei der Wahl eines geeigneten Simulationswerkzeugs müssen viele Kriterien berücksichtigt werden, wie Echtzeitfähigkeit,

Simulationsaspekte, Kollisionsbetrachtungsmöglichkeiten, Kopplung an Steuerungsplattformen etc. Mit dem Werkzeug TrySim können 3-D Kollisionsbetrachtungen und Abläufe einer Anlage sehr gut modelliert und simuliert werden. Diese Demo-Anlage in Abbildung 6.10 besteht aus einer Transportstrecke, die sekündlich (virtuelle) Päckchen produziert. Im weiteren Verlauf werden je zwei Päckchen mit Hilfe einer Overhead Chain gruppiert. Je zwei Päckchen müssen anschließend zu einem Bündel verschweißt werden, bevor die beiden ein Label aufgedruckt bekommen. Am Ende der Transportstrecke wird die korrekte Applikation des Labels geprüft und im Fehlerfall seitlich ausgeschleust. Im Erfolgsfall werden die gebündelten Päckchen nach hinten, bspw. in eine Logistikeinheit, weitergeleitet. Im Simulationssystem werden die Päckchen gezählt und am Ende virtuell zerstört.

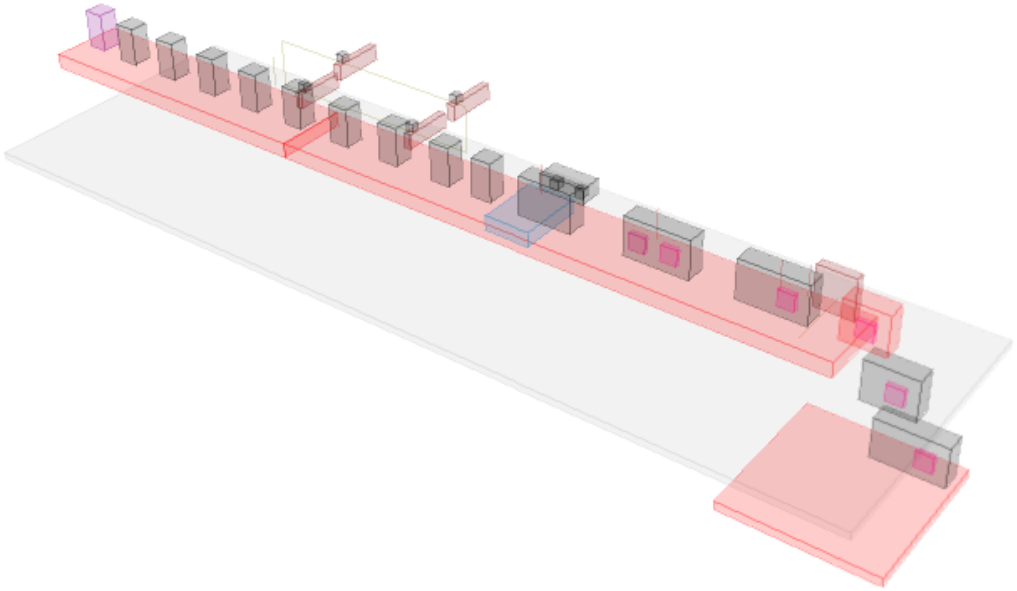


Abb. 6.10: Screenshot der 3-D Darstellung des TrySim Simulationsmodells

Bei der Qualifizierung größerer Anlagen stellen die Fehlerszenarien den größten Aufwand dar. Dabei werden verfahrens- und produktionstechnische Folgefehler häufig durch Fehler aus Defekten technischer Komponenten begründet. In dem vorliegenden Beispiel können u.a. die folgenden Fehler auftreten:

- **Overhead Chain** Bei einem Ausfall oder zeitlichem Versatz der Overhead Chain wäre die Sortierung der Päckchen gestört. Dies kann im Falle eines Stillstandes auch dazu führen, dass sich ankommende Päckchen aufstauen, was wiederum Auswirkungen auf das vorangehende Modul haben kann.
- **Heizung** Damit die Folie um die Päckchen gewickelt und verschweißt werden kann, muss diese auf die entsprechende Temperatur gebracht werden. Dieser Prozess kann abhängig von den Umgebungsbedingungen sehr sensibel sein, wodurch die Bündelung fehlerhaft werden kann.
- **Etikettierer** Mit Hilfe des Etikettierers wird ein Label auf das Bündel angebracht. Dieses kann geklebt oder auch gedruckt werden, wobei beim Druck

die Bandgeschwindigkeit und Druckgeschwindigkeit synchronisiert sein müssen, um eine entsprechende Druckqualität zu erhalten. Sofern ein Label nicht oder nur in geringer Qualität aufgebracht werden kann, muss das Bündel am Ende aussortiert werden. Damit der Etikettierer überhaupt funktionsfähig werden kann, muss der dazugehörige Sensor das ankommende Bündel korrekt erkennen können. Auch dieser Sensor kann defekt sein und somit ein korrektes Bündel nicht erkennen.

- **Ausschieber** Aus Sicht der Qualität nimmt der Ausschieber am Ende des Transportbandes und Prozesses eine zentrale Rolle ein. Die Aktorik des Ausschiebers sowie die dazugehörige Sensorik zur Erkennung des Bündels und Ausschussermittlung müssen zuverlässig sortieren. Sofern die Ansteuerung der Ausschieberplatte fehlschlägt, da bspw. Signale nicht ankommen oder eine Verklemmung vorliegt, kann es dazu führen, dass fehlerhafte Päckchen nicht aussortiert werden.

6.3.2 Fehlermodell und Steuerungsquelltext

Das Applikationsbeispiel beschreibt eine typische Teilfunktionalität in zahlreichen Branchen des Maschinen- und Anlagenbaus. So stellt bspw. die zuverlässige Sortierung von Produktionsgütern insbesondere bei Anlagen der Pharma- und Lebensmittelindustrie durch Vorgaben der FDA (Food and Drug Administration) eine nachweispflichtige Eigenschaft dar.

Für das oben beschriebene Szenario gibt es mehrere potentielle Fehlerquellen in der Beispielanlage. So kann es im Falle eines defekten Ausschiebers dazu führen, dass fehlerhafte Produkte als gut markiert und nicht ausgeschleust werden. Die Überprüfung des Anlagenverhaltens in diesem Fehlerfall ist somit ein elementares Testszenario.

Für die Modellierung des Fehlers und anschließende Evaluierung des Generierungsverfahrens wird ein defekter Ausschieber (Aktorik) als Fehler ausgewählt. Zusätzlich wird das Fehlermodell in die Simulationsumgebung TrySim integriert, um die Validität der Testdaten überprüfen zu können.

6.3.2.1 Das zustandsbasierte und parametrische Fehlermodell

Auf Basis des in Kapitel 4.2 beschriebenen Konzepts zur fehlerzentrierten Modellierung werden die möglichen Fehler des Ausschiebers in diesem Kapitel beschrieben. Zur Verdeutlichung des Konzepts werden sowohl die zustandsbasierte als auch die parametrische Beschreibung der Fehler umgesetzt. Zur Testdatengenerierung mit Hilfe der prototypischen Umsetzung dient das zustandsbasierte Fehlermodell. In Abbildung 6.11 ist die parametrische Fehlermodellierung für eine Reibungsänderung und den vollständigen Ausfall des Ausschiebers dargestellt. Dieser Zusammenhang beschreibt die Semantik der Relation, indem die Abhängigkeiten der betreffenden Parameter zueinander in Verbindung gesetzt werden.

In Abbildung 6.12 ist die zum Parameterdiagramm zugehörige Spezifikation der Constraints abgebildet. Darin sind die mathematischen Beziehungen der Parameter in Form von Constraints (Zusicherungen) spezifiziert. In dem gewählten Beispiel

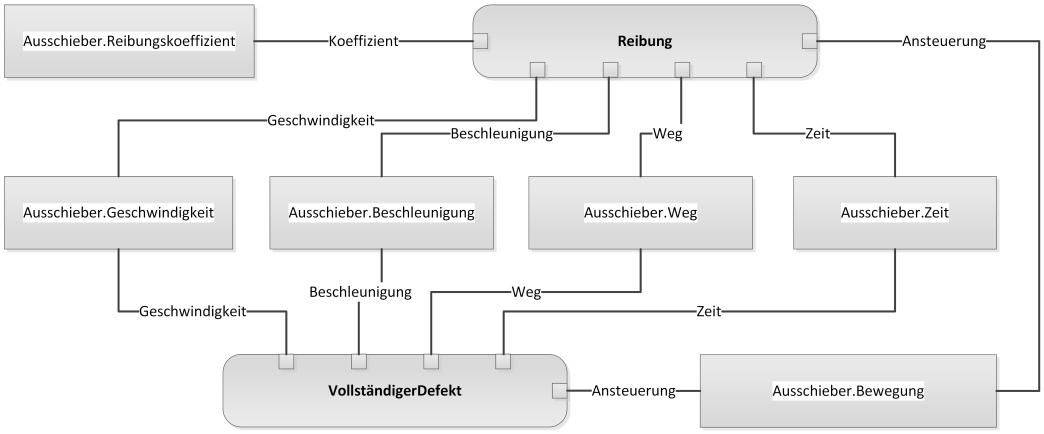


Abb. 6.11: Parametrisches Fehlermodell des Ausschiebers

lassen sich die betrachteten Fehler als funktionale Kombination der Parameter des Ausschiebers realisieren.

«constraint» **Reibung**
Constraints { Bewegung = Fahrprofil(Geschwindigkeit, Beschleunigung, Weg, Zeit, Koeffizient) }
parameters Bewegung : Position Geschwindigkeit : m/s Beschleunigung : m/s² Weg : m Zeit : s Koeffizient : const.

«constraint» **VollständigerAusfall**
Constraints { Bewegung = Fahrprofil(Geschwindigkeit, Beschleunigung, Weg, Zeit) }
parameters Bewegung : Position Geschwindigkeit : m/s Beschleunigung : m/s² Weg : m Zeit : s

Abb. 6.12: Constraints zum parametrischen Fehlermodell in Abbildung 6.11

Je nach Notwendigkeit ermöglicht die parametrische Fehlermodellierung eine sehr präzise Form der Beschreibung eines Fehlers. Die zustandsbasierte Variante lässt sich zur grobgranularen Beschreibung in frühen Phasen einsetzen. Abbildung 6.13 zeigt die Integration des Ausschieberfehlers in die Verhaltensspezifikation. Das Fehlverhalten wird als orthogonaler Zustand, d.h. als vollständig parallel ablaufender Prozess modelliert. Über die Fehlervariable kann der geforderte Fehler gesetzt werden.

Umsetzung im Simulationssystem TrySim: Durch den strukturellen Aufbau von TrySim müssen zuerst die externen Signale (I/O) als tatsächliche Schnittstelle zur SPS festgelegt werden. Dieser notwendige Modellierungsansatz folgt der grundsätzlichen Programmierlogik von Siemens Steuerungen. In einem Datenbaustein, hier DB_PLC_Interface (DB11) werden alle Ein- und Ausgangssignale inklusive der Adressen, Datentypen und einem optionalen Kommentar vergeben. Diese Schnittstellenbeschreibung ist verbindlich und muss auf der Gegenstelle (SPS) in identischer Weise eingerichtet werden. Dabei müssen lediglich die Eingabesignale zu Ausgabesignalen gesetzt werden.

Die Modellierung der gesamten Demo-Anlage, wie sie in Abbildung 6.10 dargestellt ist, erfolgt über den Elementbaum. Alle Komponenten der Anlage können hierarchisch organisiert und dadurch stark modularisiert werden. Der Ausschieber

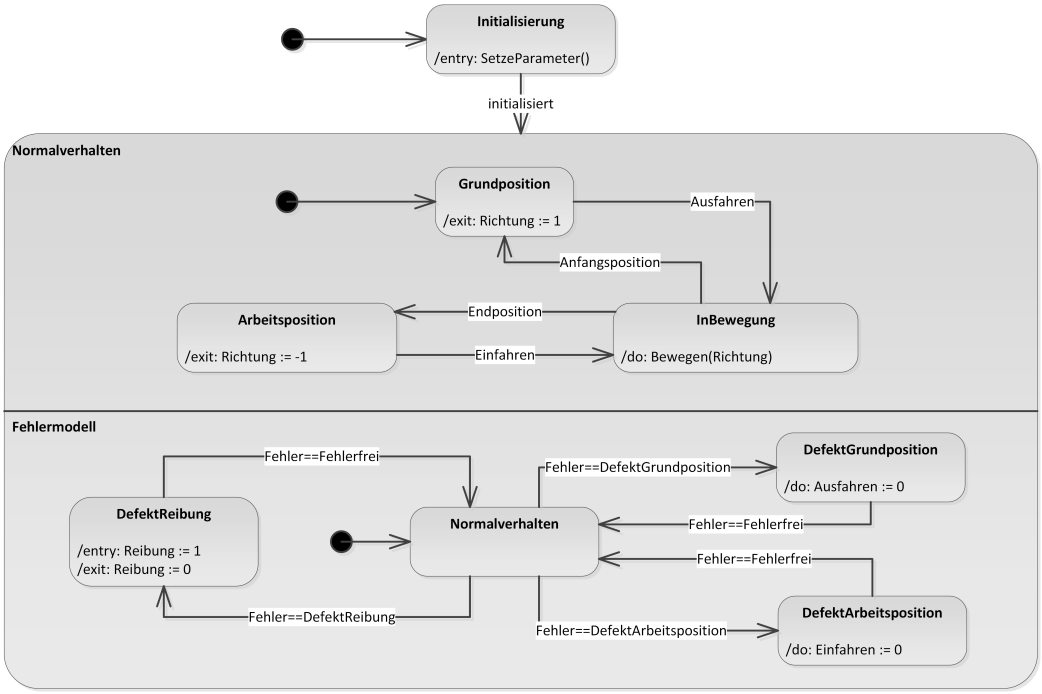


Abb. 6.13: Zustandsbasierte Beschreibung und Integration des Fehlers

wird als Linearbeweger und einem 4/2 valve Antrieb modelliert. Die Grundlage der Logikimplementierung des Simulationsmodells wird über interne Ein- und Ausgangssignale (I/O) modelliert. Dazu wird hier der Datenbaustein DB_Internal (DB13) verwendet, in dem auch die Endschalter des Ausschiebers (ES_AussGrund_Model, ES_AussArbeit_Model) festgelegt werden. Die Startparameter des Ausschiebers sind:

- Geschwindigkeit: 5 $\frac{m}{s}$
- Start-Stop-Rampe
  - Zeit: 0.2 s
  - Weg: 500 mm
  - Beschleunigung: 25 $\frac{m}{s^2}$

Im Wesentlichen werden in TrySim alle Elemente als Quader modelliert, deren physikalische Eigenschaften (Trägheit, Kollision etc.) bereits durch das Simulationssystem vorhanden sind. In einfachen Simulationsmodellen werden die externen I/O Signale (Signale zur Steuerung) direkt mit den modellierten Komponenten verbunden. Bei der Berücksichtigung von Fehlersituationen und der Möglichkeit, noch nicht vorhandene Steuerungs- oder Bedienlogik (bspw. HMI) in TrySim nachzubilden, müssen simulationsinterne Zusammenhänge und Logikabläufe implementiert werden. Dies erfolgt über entsprechende Netzwerke im Organisationsbaustein (OB 1), der in der IEC 61131-3 Programmiersprache AWL (Anweisungsliste) implementiert wird. In Listing 6.2 ist der für die Fehlermodellierung des Ausschiebers relevante Abschnitt beschrieben. Die Bewegung des Ausschiebers im Modell, die anschließend entsprechende Sensorsignale an die

angeschlossene Steuerung zurückliefert, wird nur dann aktiviert, wenn der Ausschieber nicht blockiert ist und zusätzlich das Signal DB_PLC_Interface.Output.Ausschieben von der Steuerung aus generiert wird (Zeilen 17-19 in Listing 6.2).

Listing 6.2: Interne Logik des Ausschiebers in TrySim als Netzwerk im OB 1

```
...
16: // Ausschieber InBewegung
17: A  "DB_PLC_Interface".Output.Ausschieben
18: AN "DB_Internal".Ausschieber_Blocked
19: =  "DB_Internal".Ausschieber_Model
20:
21: // Ausschieber DefektGrundposition
22: A  "DB_Internal".ES_AussGrund_Model
23: AN "DB_Internal".ES_AussGrund_ForceLow
24: AN "DB_Internal".ES_AussGrund_ForceHigh
25: =  "DB_PLC_Interface".Input.ES_AusschieberGrund
26: A  "DB_Internal".ES_AussGrund_ForceLow
27: R  "DB_PLC_Interface".Input.ES_AusschieberGrund
28: A  "DB_Internal".ES_AussGrund_ForceHigh
29: S  "DB_PLC_Interface".Input.ES_AusschieberGrund
30:
31: // Ausschieber DefektArbeitposition
32: A  "DB_Internal".ES_AussArbeit_Model
33: AN "DB_Internal".ES_AussArbeit_ForceLow
34: AN "DB_Internal".ES_AussArbeit_ForceHigh
35: =  "DB_PLC_Interface".Input.ES_AusschieberArbeit
36: A  "DB_Internal".ES_AussArbeit_ForceLow
37: R  "DB_PLC_Interface".Input.ES_AusschieberArbeit
38: A  "DB_Internal".ES_AussArbeit_ForceHigh
39: S  "DB_PLC_Interface".Input.ES_AusschieberArbeit
40:
...
52:    NOP 0
```

6.3.2.2 Der zu analysierende Steuerungsquelltext

Die Generierung der Testeingabedaten hängt einerseits von dem betrachteten Fehler, jedoch andererseits im hohen Maße von der Umsetzung der Steuerungsapplikation ab. Bei identischen Eingabeparametern können verschiedene Implementierungen einer geforderten Funktionalität durch unterschiedliche Pfade realisiert werden. Des Weiteren entscheidet auch die Berücksichtigung von Fehlerbehandlungsroutinen, ob Fehler identifiziert und Gegenmaßnahmen eingeleitet werden.

Der diesem Beispiel zugrunde gelegte Steuerungsquelltext hat insgesamt einen Umfang von 34 LOC. In Abbildung 6.14 sind die wesentlichen Zeilen des Quelltextes, die für die Testdatengenerierung maßgeblich sind, dargestellt. Der dargestellte Ausschnitt des Steuerungsquelltextes steuert den Ausschieber an, sobald ein Päckchen ohne

Etikett durch den zugehörigen Sensor (LS_Ausschieben) erkannt wurde. Die für die Bearbeitung dieses Beispiels relevanten Variablen lauten:

- **Eingabeparameter** –
- **I/O** MainPower_On (BOOL), Conveyor1_Run (BOOL), Conveyor2_Run (BOOL), Chain_Run (BOOL), LS_Melter_On (BOOL), LS_Marker_On (BOOL), LS_Ausschieben (BOOL), LS_Detect_Mark (BOOL)
- **Lokale Variablen** LS_Ausschieben_alt (BOOL), QS (BOOL)

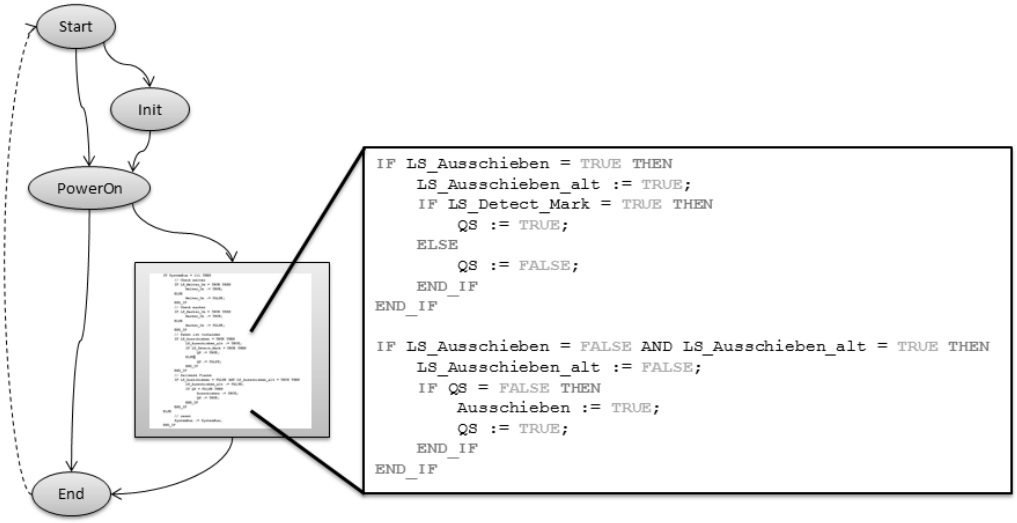


Abb. 6.14: Steuerungsquelltext zur Ansteuerung des Ausschiebers

6.3.3 Generierung der Testdaten

Bei der Auswahl des Ausschieberfehlers ergibt sich über die I/O Kopplung eine eindeutige Stelle (Anweisung) im Quelltext, für die die Testeingabedaten generiert werden sollen. Für die Testdatengenerierung in Kapitel 6.3.3.1 wird die Zeile ausgewählt, die den Ausschieber anspricht (*Ausschieben := TRUE;*).

In dem hier gewählten Beispiel werden alle gültigen Variablenwerte ermittelt, die unter Berücksichtigung der Datenabhängigkeit berechnet werden. Eine Reduktion der Wertebereiche ist bei dem gewählten Beispiel nicht notwendig, da es sich weitestgehend um boolsche Datentypen handelt. Die Bildung von Äquivalenzklassen wird wie im ersten Evaluationsbeispiel unter Abschnitt 6.2.3.2 verwendet.

6.3.3.1 Generieren der Testdaten als Simulationsparameter

In der Simulation des verwendeten Anlagenausschnitts können zahlreiche technische Komponenten defekt sein. In der Praxis werden die zu testenden Fehlerszenarien, d.h. welche technischen Komponenten fehlschlagen können, meist auf Basis der Ingenieurserfahrung selektiert und in einer Test Checkliste dokumentiert.

Abbildung 6.15 zeigt den Fehlerdialog der TrySim Simulation, über den die Fehler für das Applikationsbeispiel aktiviert werden können. Die Testdaten werden im Folgenden für einen Defekt an dem Ausschieber am Ende des Transportbandes exemplarisch ermittelt. Unter Verwendung der Simulation kann dieser Defekt über das interne Fehlermodell in der Simulation gesetzt werden, so dass der Ausschuss nicht mehr erfolgen kann, da das Funktionsverhalten überschrieben wird.

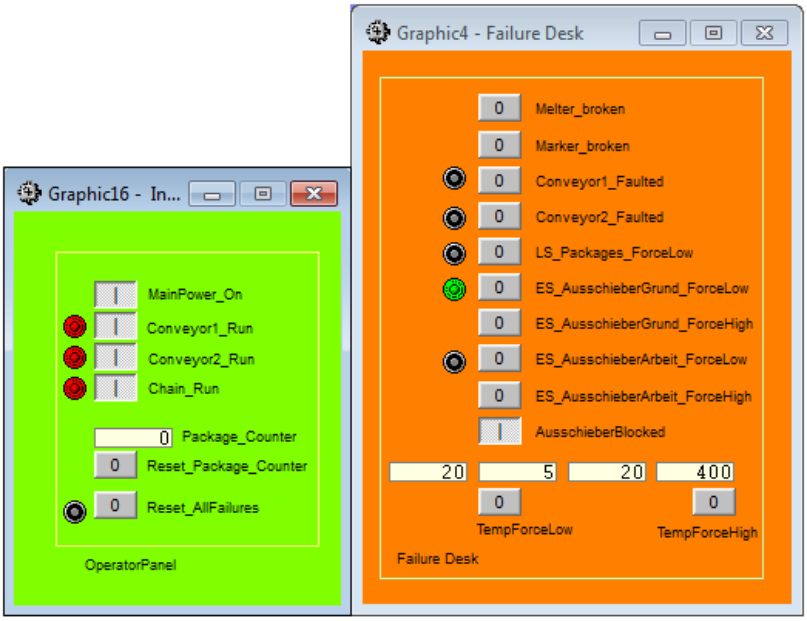


Abb. 6.15: In der Simulation hinterlegte Funktionen und forcierbare Fehler

Die Generierung der Eingabedaten für das IEC 61131-3 Programm erfolgt mit Hilfe der prototypischen Umsetzung des oben beschriebenen Verfahrens. Ein Defekt des Ausschiebers, ungeachtet der konkreten Ursache (Verklemmung, fehlender Luftdruck o.ä.), würde somit dazu führen, Ausschussware in die Logistik weiter zu transportieren. Um das Systemverhalten aufgrund der umgesetzten Steuerungslogik prüfen zu können, muss das Steuerungsprogramm mit den entsprechenden Eingabedaten (Simulationskonfiguration) stimuliert werden. Im Simulationsmodell kann der Defekt des Ausschiebers über AusschieberBlocked aktiviert werden.

Das Ergebnis der Eingabedatenberechnung ist in Tabelle 6.7 dargestellt. Mit der konkreten Wertebelegung der Eingabedaten des Steuerungsprogramms wird die in der Simulation als defekt markierte technische Komponente (Ausschieber) angesprochen. Dazu muss das System eingeschaltet werden, d.h. MainPower_On, Conveyor1_Run, Conveyor2_Run und Chain_Run müssen aktiv sein. Diese Werte können über den Operator Panel, siehe Abbildung 6.15 links, gesetzt werden. Aufgrund der Implementierung des Steuerungsprogramms können der Bündler (Melter_On) als auch der Etikettierer (Marker_On) sowohl funktional als auch fehlerhaft sein, da diese beide Werte (0,1) annehmen können. Damit der Ausschieber angesprochen wird, muss LS_Ausschieben seinen Wert von 1 auf 0 (fallende Flanke) ändern.

Die Betrachtung der Performance des Verfahrens ist in Tabelle 6.8 zusammengefasst. Die Erstellung des Kontrollflussgraphen erfolgt in einer sehr kurzen Zeitdauer und kann mit ca. 16 ms in diesem Evaluationsbeispiel quantifiziert werden. Die Formulierung

Tab. 6.7: Ergebnis der Eingabedatenberechnung

Eingabevariable	**Werte$_1$**	**Werte$_2$**
MainPower_On	1	1
Conveyor1_Run	1	1
Conveyor2_Run	1	1
Chain_Run	1	1
LS_Melter_On	{0, 1}	{0, 1}
LS_Marker_On	{0, 1}	{0, 1}
LS_Ausschieben	1	0
LS_Detect_Mark	0	0

bzw. Ableitung des Constraint Satisfaction Problems erfolgt als Analyseergebnis des Kontrollflussgraphen in ca. 40 ms. Den stärksten Umfang der gesamten Testdatengenerierungskette vereinnahmt die eigentliche Berechnung, ist jedoch mit ca. 340 ms bzw. ca. 20 ms schnell abgeschlossen. Das Parsen der auf der Standardausgabe ausgegebenen Ergebnisse erfolgt derart schnell, dass der Vorgang bei einigen Zeitmessungen nicht erfasst werden konnte und im Mittel mit 0,2 ms gemessen wurde. Die Verarbeitung der Ergebnisse ist bei der Auswertung von einem generierten Datensatz ebenfalls nicht aufwändig, so dass die gesamte Berechnungsdauer ca. 400 ms bzw. ca. 80 ms beträgt.

Tab. 6.8: Ergebnisse der Performance des Evaluationsbeispiels für die einzelnen Prozessschritte der Testdatengenerierung

Prozessschritt	**Berechnungsdauer**	
	Datenabh.	KLEE
CFG erstellen	15,7 ms	
CSP formulieren	41 ms	
Lsg berechnen	344 ms	20,2 ms
Ausgabe parsen	0,2 ms	0,0 ms
Ergebnis verarbeiten	4,4 ms	4,6 ms
Summe	**405,3** ms	**81,5** ms

Im vorangegangenen Fall wurde ein gültiger Testeingabedatensatz erzeugt. Die Wertebelegungen der Eingabedaten entsprechen in diesem Fall dem ersten gültigen Wert, der alle Constraints des CSPs erfüllt. Zur Ermittlung des ersten gültigen Wertes, wird der jeweils kleinste Wert aus dem Wertebereich der korrespondierenden Variablen des CSPs verwendet. Somit stellt der einzelne Datensatz einen Grenzwert zur Stimulation des Testobjekts dar, siehe Kapitel 3.4.1.1. Die Generierung aller gültigen Eingabewerte stellt einen komplexeren Berechnungsaufwand dar, liefert jedoch die vollständigen Werte der Eingabedaten zur anschließenden Stimulation. Durch die im Konzept integrierte automatische Äquivalenzklassenbildung werden die ermittelten Intervalle über alle Eingabeparameter (CSP-Variablen) miteinander kombiniert, so dass aus diesen gültigen Wertekombinationen beliebige Repräsentanten ausgewählt werden können. Zur Berechnung der Testeingabedaten ist keine weitere Angabe von Parametern aufgrund der Datenabhängigkeit notwendig. Die Generierung berücksichtigt

automatisch alle existierenden def-use Abhängigkeiten zwischen lokalen Variablen und erzeugt zusätzliche CSPs, die in umgekehrter Reihenfolge berechnet werden.

6.3.3.2 Zusätzliche Betrachtung der Sensor Aktor Relation

In Abschnitt 4.4.4 ist das Konzept der Berücksichtigung von Sensor Aktor Funktionseinheiten erläutert. Anhand dieses Evaluationsbeispiels lässt sich der Zusammenhang zwischen dem Ausschieber als Aktor und dessen Sensor zur Erkennung des Markers eines Päckchens (LS_Detect_Mark ist auf low, d.h. 0) erschließen. Dies bedeutet, dass jedes ankommende Päckchen aussortiert werden muss, zu dem der zugehörige Sensor das low Signal liefert bzw. falls der Sensor defekt ist und permanent ein low Signal anliegt. Für die unmittelbar gekoppelten Funktionseinheiten aus Sensor und Aktor, deren Ansprechverhalten nicht abhängig ist von Querbezügen wie bspw. dem Prozess auf einer Anlage, können die Testeingabedaten unmittelbar ermittelt werden. Das beschriebene Verfahren zur automatischen Berechnung der Eingabedaten vereinfacht diesen Vorgang.

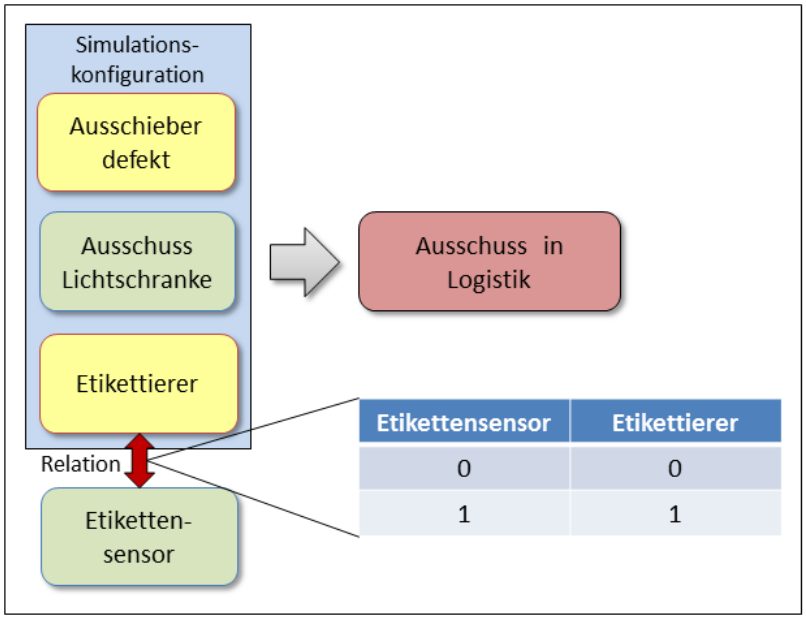


Abb. 6.16: Berücksichtigung prozessualer Abhängigkeiten von Sensor und Aktor

In Abbildung 6.16 ist eine weitere mögliche Sensor Aktor Relation im Rahmen des Evaluationsbeispiels dargestellt. Das Berechnungsergebnis der Simulationsparameter verdeutlicht, dass der Etikettensensor defekt gesetzt werden muss. Aufgrund der prozessualen Abhängigkeit zwischen dem Etikettierer (Aktor) und dem späteren Etikettensensor, kann das Ansteuern des (defekten) Ausschiebers auch dadurch ermöglicht werden, indem der Etikettierer defekt und der Sensor funktional gesetzt werden. Unter Berücksichtigung dieser einfachen Relation werden weitere, ebenfalls gültige Testeingabedaten ermittelt, wodurch die Testtiefe entsprechend erhöht wird. Durch diese Erweiterung des Generierungsverfahrens können prozessbezogene Zusammenhänge mit betrachtet werden. Die berechneten Parameter (I/O Wertebelegungen) repräsentieren die Simulationszustandskonfiguration.

6.3.4 Zusammenfassung

In diesem Evaluationsbeispiel wurden mögliche Fehler technischer Komponenten einer Verpackungsanlage in ein SysML Fehlermodell überführt. Die Umsetzung des Modells erfolgte einerseits in der grobgranularen zustandsbasierten Form als auch in der präzisen parametrischen Form. Für die Testdatengenerierung mit Hilfe der prototypischen Umsetzung wurde das zustandsbasierte Fehlermodell, isoliert vom restlichen Gesamtmodell, in TrySim integriert und an den Steuerungsquelltext über die I/O Kopplung angebunden. Die Testdatengenerierung hat die fürs Ansprechen des Fehlers notwendigen Stimuli des Testobjekts und somit die Parameter der Simulationsumgebung generiert. Die Berechnungskomplexität liegt unter 1 s und erfolgt damit instantan. Im Folgenden wird das Evaluationsbeispiels im Hinblick auf die gestellten Anforderungen aus Kapitel 2.4.2 betrachtet.

- **A1** Die Testdatengenerierung für permanente, mechanische Fehler ist bereits gemäß dem erstellten Konzept realisiert. In dem betrachteten Beispiel wurden Testeingabedaten generiert, die bei der Stimulation des Testobjekts und aktiviertem Fehlerzustand (gemäß Fehlermodell) die Funktionalität ansprechen, die einen Fehler nach sich zieht.

- **A2** Die Testdaten werden automatisch generiert. Insbesondere können durch die Berücksichtigung der Datenabhängigkeit Testeingabedaten für zyklisch aufgerufene Funktionseinheiten generiert werden. Nach der Auswahl des potentiellen Fehlers im Fehlermodell war in dem Evaluationsbeispiel eine eindeutige Zuordnung im Steuerungsquelltext möglich. Die aktuelle Umsetzung ermöglicht eine gezielte Skalierung des Umfangs der Testdatengenerierung.

- **A3** Die im Konzept integrierte, zweigliedrige Fehlermodellierung unterstützt einerseits eine grobgranulare Berücksichtigung in frühen Phasen und ermöglicht andererseits eine präzise Detaillierung des Fehlers in Form einer parametrischen Darstellung. Zur Evaluation wurde die zustandsbasierte Variante verwendet.

- **A4** Ein wesentliches Merkmal des Konzepts der Testdatengenerierung stellt die Reduktion der Eingabedaten auf eine repräsentative Menge dar. Dafür wurde ein Algorithmus zur automatischen Äquivalenzklassenbildung im Konzept eingeführt und im Prototypen realisiert. In diesem Evaluationsbeispiel gab es durch die eingeschränkten Datentypen (überwiegend bool) exakt eine Äquivalenzklasse. Die Messungen der Performance belegten des Weiteren die Effizienz des Verfahrens.

- **A5** Der Zusatzaufwand der Fehlermodellierung muss als Teil der Systemmodellierung betrachtet und kann nicht allgemein bewertet werden. Da die Definition eines Fehlers gemäß Kapitel 3.2 als eine Abweichung eines Bezugswertes oder Erwartungswertes zu betrachten ist, obliegt es dem Modellierer, welche Aspekte als Fehler betrachtet werden. Die in diesem Evaluationsbeispiel berücksichtigten Fehler stellen lediglich einen Bruchteil des gesamten Applikationsfunktionsumfangs dar. Die Testdatengenerierung stellt quasi keinen Aufwand dar, da sie, mit Ausnahme der Fehlerauswahl, vollautomatisch abläuft und die Ergebnisse aggregiert.

6.4 Testdatengenerierung aus anderen IEC 61131-3 Sprachen

In den Evaluationsbeispielen wurde als Ausgangsquelltext Strukturierter Text (ST) eingesetzt. Durch die funktionale innere Überführbarkeit aller IEC 61131-3 Sprachen kann aufgrund der Transformationsregeln jeder Ausgangsquelltext in ST überführt werden und das Konzept der Testdatengenerierung grundsätzlich angewendet werden. Bei genauer Betrachtung gängiger Quelltexttransformation wird deutlich, dass der aus den graphischen IEC 61131-3 Sprachen, als auch aus AWL transformierte ST Zwischenquelltext keiner intuitiven Programmierweise entspricht, sondern den starren Regeln der jeweiligen Ausgangssprache folgt. Bei der Transformation von AWL in ST werden Bedingungen, die anschließend zu Sprüngen im Ablauf führen, nicht als *IF*-Bedingungen dargestellt, sondern ebenfalls in Form von spezifischen Sprunganweisungen repräsentiert. Im Falle von graphischen Programmiersprachen werden u.a. auch definierte Ablaufstrukturen generiert, deren sequentielle Abarbeitung dem Funktionsumfang der Ausgangssprache entsprechen.

Der Funktionsumfang der prototypischen Implementierung umfasst die Testdatengenerierung auf dem ST Quelltext für Steuerungssoftware. Durch die spezifischen Transformationsregeln der restlichen IEC 61131-3 Sprachen auf ST müssen zur Realisierung der Testdatengenerierung die strukturellen Spezifika zur Logikabbildung bei der Datenabhängigkeitsberechnung und der anschließenden CSP-Formulierung berücksichtigt werden. Die restlichen Prozessschritte der prototypischen Implementierung entlang der Generierungskette können unverändert eingesetzt werden.

6.5 Gesamtbewertung von Konzept und Umsetzung

In dem vorliegenden Kapitel erfolgte eine Evaluation des Konzepts und der prototypischen Implementierung. Dabei hat sich herausgestellt, dass die beschriebene Methodik, die Modellierung von permanenten, mechanischen Fehlern ermöglicht und somit eine geeignete Grundlage zur automatischen Testdatengenerierung darstellt. Die Erfüllung der Anforderungen wurde in den Kapiteln 6.2.4 und 6.3.4 der Evaluationsbeispiele ausführlich erläutert. Durch die Unterstützung einer zweigliedrigen Fehlermodellierung können Fehler in unterschiedlicher Granularität, abhängig von der Phase des Entwicklungsprozesses, beschrieben werden.

Des Weiteren werden Fehler im Rahmen des präsentierten Konzepts als relative Größe betrachtet, so dass der Umfang der Fehlermodellierung skalierbar ist. Insbesondere bei der zustandsbasierten Fehlermodellierung entfällt ein vernachlässigbar geringer Zusatzaufwand, der über die Spezifikation des Normalablaufs hinausgeht. Bei einer möglichen weiteren Verwendung des Systemmodells als Teil einer Simulation sind abstrakte Beschreibungen meist in Form von zustandsbezogenen Modellen üblich. Dadurch können zustandsdiskrete Fehlermodelle unmittelbar integriert werden und stellen somit einen zusätzlichen Mehrwert in der gesamten Entwicklung dar, wodurch sich der initial erhobene Mehraufwand relativiert.

Die hohe Automatisierung der Testdatengenerierung, nach Auswahl des zu betrachteten Fehlers, erfolgt in kurzer Berechnungszeit, was durch die jeweiligen Messungen der Performance belegt wurde. Der Großteil der Berechnungskomplexität fällt dabei

auf das Lösen des Constraint Satisfaction Problems. Die Berechnung einer Lösung kann mit dem KLEE Verfahren in sehr kurzer Zeit ermittelt werden. Die in diesem Konzept eingeführte Berücksichtigung der Datenabhängigkeit ermöglicht die Generierung von Eingabedaten für zyklische IEC 61131-3 Softwarebausteine, da durch die Ausführungslogik ausgewählte Zweige nur nach vorheriger Ausführung abhängiger Zweige erreicht werden kann. Das Konzept der Datenabhängigkeit wurde im Evaluationsbeispiel erfolgreich bestätigt, wodurch mehrere zeitdiskrete Eingabedatensätze bei beiden Verfahren generiert werden.

Bei der Ermittlung aller möglichen Eingabedaten werden durch den Constraint Solver alle gültigen Variablenwerte berechnet. Durch einen eingeführten Algorithmus werden die Strukturinformationen der Lösungen analysiert und automatisch Äquivalenzklassen erzeugt, denen alle weiteren Lösungsvektoren mit einem einfachen geometrischen Abstand hinzugefügt werden. Somit werden automatisch Gruppen von Äquivalenzklassen erzeugt, deren Mächtigkeit einen Hinweis auf die Relevanz der jeweiligen Eingabeparameter liefert.

Der quantitative Vergleich der beiden Verfahren zeigt die charakteristischen Eigenschaften der Berechnungsvorschriften. Die Ergebnisse lassen sich wie folgt zusammenfassen:

- Das Datenabhängigkeitsverfahren kann sowohl eine Lösung als auch alle möglichen Variablenwerte berechnen, wohingegen mit dem KLEE Verfahren nur eine Lösung ermittelt werden kann.
- Die Berechnung einer Lösung erfolgt im KLEE Verfahren deutlich schneller.
- Die Bildung von Äquivalenzklassen reduziert mögliche Testvektoren stark.
- Unter Verwendung der Datenabhängigkeitsinformation kann die Anzahl der Aufrufe für das KLEE Verfahren verwendet und somit die Berechnung beschleunigt werden.
- Die Berechnung aller Lösungen ist bis zu einem Parameterraum von 2^{16} möglich.

Im Kapitel 6.4 wird die Erweiterbarkeit auf die restlichen IEC 61131-3 Sprachen thematisiert. Dabei wurde deutlich, dass eine derartige Unterstützung des Sprachumfangs dadurch erreicht wird, indem die Logikabbildungen der Sprachen auf den Zwischenquelltext im Analysevorgang (Kontrollflussgraph etc.) berücksichtigt werden muss. Die Prozessschritte der Berechnung, Ergebnisauswertung und etwaiger Äquivalenzklassenbildung bleiben, durch den sequentiellen Ablauf der Generierungskette, davon unbeeinflusst.

Im Folgenden werden das Konzept und die Implementierung gegen die in Kapitel 3.1.1 und 3.1.2 abgeleiteten Kriterien bewertet:

- **/S1** Durch die Fehlermodellierung in SysML und die Integration des Fehlermodells in das drei Sichten Gesamtmodell werden unterschiedliche Sichten auf das System unterstützt.

- **/S2** Das Konzept unterstützt eine zweigliedrige Fehlermodellierung in Form einer parametrischen und einer zustandsbasierten Beschreibung. Dadurch wird für unterschiedliche Phasen der Entwicklung eine der Informationstiefe entsprechende Modellierungsunterstützung realisiert. Grob spezifizierte Fehler können im Laufe der Entwicklung und dem damit einhergehenden Erkenntnisgewinn mit dem SysML Parameterdiagramm präzise formuliert werden.
- **/S3** Durch die Verwendung der SysML werden die branchenspezifischen Anforderungen einer graphischen Modellierungssprache erfüllt, wodurch die Akzeptanz des Verfahrens ausdrücklich gefördert wird.
- **/S4** Das Fehlermodell ist derart konzipiert, dass es sich nahtlos in das Gesamtmodell integriert. Dadurch kann das Fehlermodell für Applikationen, die über die fehlerbasierte Testdatengenerierung hinausgehen, eingesetzt werden.
- **/S5** Das Drei-Sichten-Konzept ist insbesondere durch die Sicht des technischen Prozesses unmittelbar für den Maschinen- und Anlagenbau geeignet. Die Erweiterung um das Fehlermodell ist für den Maschinen- und Anlagenbau konzipiert, jedoch nicht darauf beschränkt.
- **/S6** Die im Speziellen bei der zustandsbasierten Fehlermodellierung angewandte Semantik ist auf die zyklische Ausführungslogik von IEC 61131-3 basierten Systemen hin konzipiert.
- **/T1** Das Fehlermodell im Gesamtsystemmodell dient als Grundlage der Testdatengenerierung. Dadurch werden Testdaten speziell für fehlerbehaftete Komponenten des technischen Systems erzeugt.
- **/T2** Durch den Algorithmus zur Bildung von Äquivalenzklassen und die Einschränkung des Parameterraumes mit Hilfe zahlreicher Constraints werden die Testdaten zur Berechnungszeit reduziert.
- **/T3** Die Testdatengenerierungskette ist derart automatisiert, dass nach der anfänglichen Auswahl des zu berücksichtigten Fehlers der Kontrollflussgraph erstellt, die CSPs mit automatischer Datenabhängigkeitsanalyse formuliert und anschließend berechnet werden. Bei der vollständigen Variablenwertberechnung werden zusätzlich Äquivalenzklassen gebildet, so dass das Kreuzprodukt über alle Eingabeparameter gültige Testeingabedaten ergibt. Die Quelltexttransformation in die Programmiersprache C für die anschließende Verarbeitung mit KLEE erfolgt vollständig automatisiert.
- **/T4** Die bisher existierenden Ansätze zur Testdatengenerierung waren für ereignisbasierte Systeme konzipiert. Durch die zyklische Ausführungslogik können Pfade der Software teilweise erst durch mehrfache Stimulation erreicht werden. Mit dem präsentierten Konzept existiert eine Lösung zur automatischen Berücksichtigung der Datenabhängigkeit bei der Berechnung zeitdiskreter Eingabedaten. Die Erfüllung dieses Kriteriums wurde durch das Evaluationsbeispiel in diesem Kapitel belegt.

- **/T5** Das Konzept der Testdatengenerierung basiert auf einer White-Box Analyse, wodurch die tatsächlich vorhandene Umsetzung analysiert wird. Das strukturorientierte Verfahren berücksichtigt dabei das intendierte Verhalten der Steuerungssoftware und kann sowohl nicht erreichbare Anweisungen identifizieren als auch den Abdeckungsgrad von Anweisungen und Zweigen der analysierten Softwarefragmente ermitteln.

- **/T6** Der hohe Automatisierungsgrad und die Performance des Verfahrens belegen den geringen Aufwand der Testdatengenerierung. Zur Reduktion der Ausführungskomplexität wird eine einmalige Eingrenzung der Wertebereiche notwendig. Eine Weiterentwicklung des Prototypen bietet zusätzliches Potential in der Erhöhung der Benutzerfreundlichkeit und Aufwandsreduktion, indem Wertebereiche automatisch analysiert und extrahiert oder aus Planungsdokumenten eingelesen und verwendet werden können.

- **/T7** Durch die starke Kopplung an Fehler des technischen Systems werden gezielt für den ausgewählten Aspekt Testdaten generiert. Im Vergleich zu existierenden Ansätzen bietet das Konzept dadurch eine Erhöhung der Aussagekraft der generierten Testeingabedaten.

Aus den in diesem Kapitel gewonnenen Erkenntnissen lässt sich eine Verallgemeinerung des Konzepts erschließen. Die Fehlermodellierung wurde gemäß den Anforderungen speziell für permanente, mechanische Komponentenfehler konzipiert. Mit der aufgestellten Modellierungsmethodik können letztendlich sämtliche Fehler abgebildet werden, deren Auswirkung sich logisch darstellen lässt. So können bspw. auch abrupte Ausfälle der Stromversorgung derart modelliert werden, dass bei nicht vorhandener USV (Unterbrechungsfreie Stromversorgung) die Komponente elektrisch nicht mehr versorgt und somit unmittelbar die Funktionserfüllung beendet wird. Dies kann auf Ebene der Zustandsdiagramme durch eine sog. Exception Transition in einen Zustand der Funktionsinaktivität modelliert werden. Erweiterungen der Fehlermodellierung um sporadisch, nicht systematisch auftretende Fehler lassen sich nur eingeschränkt verallgemeinern. Häufig liegen die Daten der Auftretensgründe auch nicht vollständig vor, so dass deren exakte Modellierung nicht möglich ist. Eine Möglichkeit, sporadische Fehler nachzubilden, ist, diese als permanente Fehler zu modellieren und diese bei Bedarf – bspw. in einer möglichen Simulation – einmalig auftreten zu lassen, um das Systemverhalten für diese Situation zu beobachten.

Bei Tests von Fehlerszenarien in industriellen Systemen werden meist vollständige Defekte und keine schleichenden oder zufälligen Defekte geprüft. Derartige Fehlermodelle beinhalten die Möglichkeit, Komponenten vollständig defekt zu setzen, ohne ein präzises Komponentenverhalten im Fehlerfall zu beschreiben. Häufig ist dies in der Realität auch gar nicht bekannt. Das beschriebene Verfahren zur Modellierung von Fehlern unterstützt beide Formen der Verhaltensmodellierung.

Die Integration des Fehlermodells als Datengrundlage in den Prozess der Testdatengenerierung verfolgt das Ziel, Testdaten einer höheren Aussagekraft zu erzeugen. Aus rein technischer Sicht, ist die Fehlerbetrachtung kein notwendiges Kriterium zur Testdatengenerierung. Demnach kann der Generierungsansatz um zusätzliche Aspekte erweitert und so für beliebige Anweisungen und Testeingabedaten erzeugt werden. Das

KLEE Verfahren kann zusätzlich dazu eingesetzt werden, um Abdeckungsanalysen, siehe Kapitel 3.4.2, für IEC 61131-3 basierte Applikationen durchführen.

6.6 Zusammenfassung

In diesem Kapitel wurde sowohl das Konzept als auch die prototypische Implementierung der automatischen Testdatengenerierung zur Absicherung fehlerbehafteter reaktiver Automatisierungssysteme evaluiert. Dabei wurde einerseits das Potential der Fehlermodellierung in SysML als Teil des Gesamtmodells vorgestellt sowie die Leistungsfähigkeit der automatischen Generierung von Testeingabedaten für zyklische Softwarebausteine, als Lösung von Constraint Satisfaction Problemen. Neben den Praxisbeispielen wurden weitere Erkenntnisse über die Erweiterbarkeit und Allgemeingültigkeit in diesem Kapitel erörtert und die Erfüllung der Anforderungen und Bewertungskriterien aus den Kapiteln 2 und 3 begründet.

KAPITEL 7

Zusammenfassung und Ausblick

In dieser Arbeit wurde basierend auf den erhobenen Anforderungen ein Testdatengenerierungsverfahren zur Absicherung fehlerbehafteter reaktiver Systeme entwickelt. Im vorangegangenen Kapitel erfolgte eine Bewertung des Konzepts und der prototypischen Implementierung anhand zweier praxisrelevanter Evaluationsbeispiele. Abschließend erfolgt in diesem Kapitel eine Kurzzusammenfassung der Problemstellung und des erarbeiteten Konzepts. Identifizierte Anknüpfungspunkte für zukünftige Erweiterungen des Konzepts runden die Arbeit ab.

7.1 Ergebnisse und Beitrag der Arbeit

In der Motivation zu dieser Arbeit und der anschließenden Problemanalyse wurde die Bedeutung des Tests verdeutlicht. Bei der Betrachtung der Ist-Situation im Maschinen- und Anlagenbau wurde deutlich, dass der Test, insbesondere der Softwaretest, meist keinen wesentlichen Bestandteil des Entwicklungsprozesses darstellt, sondern als Teil der Softwarekonstruktion betrachtet wird. Durch den hohen Zeit- und Kostendruck in der Projektabwicklung wird meist für die Testdurchführung nicht genügend Zeit eingeräumt. Der enge verfügbare zeitliche Rahmen zum Softwaretest wird primär zur Verifikation des Geradeauslaufs verwendet. Da reaktive Systeme den physikalischen Einflüssen aus ihrer Umwelt ausgesetzt sind, ist die steuerungs- und regelungstechnische Reaktion auf extern in das System eingetragene Fehler eine wesentliche Aufgabe der Softwarefunktionalität. Zur Erhöhung der Absicherung fehlerbehafteter Systeme im Maschinen- und Anlagenbau werden zeiteffiziente Verfahren benötigt, so dass diese im Rahmen der Entwicklung eingesetzt werden können.

In dieser Arbeit wurde ein Verfahren zur Ermittlung zeitdiskreter Testdaten für SPS Softwarebausteine entworfen, die sich in Interaktion mit potentiell fehlerhaften Komponenten befinden. Dazu wird im Rahmen der modellbasierten Systementwicklung ein auf SysML basiertes Fehlermodell vorgeschlagen, welches als Grundlage für die Testdatengenerierung für zyklische IEC 61131-3 Softwarebausteine eingesetzt wird.

Der Kern des Verfahrens ist die Transformation des Ausgangsquelltextes in eine Zwischenrepräsentation, aus der die Ermittlung der Testeingabedaten als Constraint Satisfaction Problem (CSP) formuliert wird. Auf dieser Zwischenrepräsentation erfolgt eine Analyse der Datenabhängigkeit, deren Resultat die für die Erreichbarkeit der

zyklisch ausgeführten Bausteine notwendig ist. Unter Ausnutzung dieser Zusammenhangsinformation kann der Kontrollflussgraph ausgerollt werden. Die Auswahl des Testkriteriums wird unter der Einbeziehung der Fehlermodellinformationen und einer bidirektionalen Kopplung mit dem zu analysierenden IEC 61131-3 Quelltext vorgenommen. Die beiden entwickelten Verfahren nutzen die konkreten Relationen zwischen Teilpfaden des analysierten Quelltextes. Im Falle des Verfahrens mit der direkten Abhängigkeitsbetrachtung (Abschnitt 4.4) werden genau so viele CSP Probleme erzeugt und gelöst, dass die Eingabedaten zur Erreichbarkeit berechnet werden können. Mit dem KLEE Verfahren (Abschnitt 4.5) werden in dem transformierten Quelltext exakt so viele Aufrufe des in C überführten Funktionsblocks generiert, dass die Erreichbarkeit der ausgewählten Anweisung berechnet werden kann. Beide Verfahren setzen auf einen Constraint Solver Mechanismus zur Berechnung der Eingabedaten zur Stimulation des Testobjekts. Das KLEE Verfahren ist dabei besonders schnell in der Berechnung eines einzelnen Testdatensatzes, wohingegen mit dem Datenabhängigkeitsverfahren alle Wertebelegungen der Variablen innerhalb des Wertebereiches der Parameter ermittelt werden können, die denselben Pfad der Ausführung liefern. Die Bildung von Äquivalenzklassen aus den berechneten Lösungen reduziert dabei den Lösungsraum nochmal derart stark, dass für den anschließenden Test lediglich Repräsentanten aus der jeweiligen Klasse ausgewählt werden müssen.

Die Umsetzung des Konzepts ist im Speziellen auf CoDeSys basierte Speicherprogrammierbare Steuerungen ausgelegt. Darin werden alle IEC 61131-3 basierten Bausteine der Steuerungsapplikation auf einen gemeinsamen Zwischenquelltext in Strukturiertem Text (ST) abgebildet, was das entwickelte Verfahren grundsätzlich für alle eingebundenen Programmiersprachen ermöglicht. Die beiden folgenden zentralen Aspekte dieser Arbeit wurden getestet:

- **Fehlermodell** Differenzierende Qualitätsmerkmale eines Gesamtsystems wie Stabilität und Zuverlässigkeit werden maßgeblich vom kontrollierten Systemverhalten im Fehlerfall beeinflusst. Fehler stellen ein relatives Maß dar und sind als Abweichung einer Erwartungshaltung definiert, siehe Kapitel 3.2. Das entwickelte Fehlermodell bietet die Möglichkeit, Fehler, d.h. Abweichungen vom Normalverhalten, in geforderter Granularität zu modellieren und in das Systemmodell zu integrieren. Für die Evaluation wurde die zustandsbasierte Verhaltensbeschreibung integriert.

- **Testdatengenerierung** Da die Funktionsrealisierung durch die Steuerungsapplikation erbracht wird, hängt die Qualität des Gesamtsystems im Fehlerfall hochgradig von der Steuerungslogik ab. Die Generierung übernimmt alle relevanten Schritte von der Quelltextanalyse über die Formulierung des Constraint Satisfaction Problems und Transformation in die Programmiersprache C bis zur Auswertung der Berechnungsergebnisse.

In Anbetracht der erhobenen Forderungen stellen die im Folgenden beschriebenen Merkmale den wesentlichen wissenschaftlichen und technischen Beitrag dieser Arbeit dar.

Anforderungen im Maschinen- und Anlagenbau: Die Anforderungen an Methoden und Verfahren im Maschinen- und Anlagenbau lassen sich mit Pragmatik, Effizienz und hoher Anwenderorientierung subsumieren. Die Erstellung des Konzepts dieser Arbeit wurde kontinuierlich an den Branchenanforderungen gespiegelt. In Kapitel 6 ist in der Evaluation des Konzeptes die Erfüllung der Anforderungen detailliert beschrieben.

Im Maschinen- und Anlagenbau sollten Modellierungssprachen graphisch sein, da diese die aufgrund der Branchenspezifika höchste Akzeptanz erfahren, siehe Abschnitt 3.1.1. In dieser Arbeit wurde ein Konzept zur Integration des Fehlermodells in das Drei-Sichten-Konzept erarbeitet. Als graphische Beschreibungssprache dient hierzu die SysML. Die Generierung von Testeingabedaten erfolgt mit minimalem Aufwand, da alle Berechnungsschritte vollautomatisch ablaufen.

Zweistufige Fehlermodellierung: In der Entwicklung komplexer Systeme erfolgt zur Komplexitätsreduktion eine Zerlegung in beherrschbare Teilsysteme, die einzeln betrachtet und definiert werden können. Häufig verfeinern sich die Erkenntnisse entlang des Entwicklungsprozesses so, dass anfänglich grobe Beschreibungen sukzessive detailliert werden. Bei der Analyse und Beschreibung von Fehlern eines Systems wird sich dieser Erkenntnisgewinn noch stärker bemerkbar machen, da der anfängliche Fokus auf der Primärfunktionalität liegt. Aus diesem Grund unterstützt das entwickelte Konzept eine zweistufige Modellierung für Fehler. In frühen Phasen kann die zustandsdiskrete Beschreibungsform zur abstrakten und grobgranularen Beziehungsformulierung eingesetzt werden. Im weiteren Verlauf bietet die parametrische Darstellung in Form des SysML Parameterdiagramms die Unterstützung, die mathematischen Zusammenhänge zwischen Systemelementen und Umwelt präzise zu modellieren und in das Systemmodell zu integrieren. Der tatsächliche Bedarf der Beschreibungsgranularität von Fehlern hängt im starken Maße vom Anwendungsgebiet des modellierten Systems, der Fehlerart und von der eingesetzten Testumgebung ab.

Testdaten für zyklische Programme: Die Testdatengenerierung dieser Arbeit kombiniert die Forderung einer hohen Aussagekraft der Testdaten durch den direkten Bezug zu Komponentenfehlern. Dabei wird die zyklische Ausführungslogik durch eine Datenabhängigkeitsanalyse disjunkter Pfade berücksichtigt. Im Vergleich zu existierenden Testdatengenerierungsansätzen, siehe Kapitel 3, erfolgt mit dem entwickelten Verfahren eine zielgerichtete Reduktion der Testdaten. Das Konzept grenzt sich insbesondere durch die Berechnung und Integration der Datenabhängigkeit in das Generierungsverfahren ab, da somit zeitdiskrete Eingabedaten durch Stimulation des Testobjekts ermittelt werden können. Existierende Verfahren können aufgrund von Pfadabhängigkeit durch einmalige Ermittlung der Stimuli keine gültigen Eingabesequenzen ableiten. Die mehrfache Auswertung durch ein Ausrollen des Kontrollflussgraphen ermöglicht die Berechnung des konkreten Pfades. Eine zusätzlich eingeführte Berechnungsvorschrift ermittelt aus den errechneten Lösungen automatisch Äquivalenzklassengruppen, deren Mächtigkeit eine zusätzliche Aussagekraft der Güte zulassen.

7.2 Einordnung des Konzepts

In Kapitel 3 erfolgte eine Bewertung existierender Ansätze zur Modellierung und Testgenerierung. Mit der vorliegenden Arbeit wurde ein Konzept zur Modellierung und Integration von Fehlern in das Systemmodell zur Erfüllung der Anforderungen im Maschinen- und Anlagenbau entwickelt. Dadurch grenzt sich diese Arbeit von den bisherigen Konzepten ab, indem die Lücke der Fehlerberücksichtigung geschossen wird.

Der zweite Teil der Arbeit betrachtet die Testdatengenerierung für IEC 61131-3 Programme. Bei vergleichbaren Ansätzen aus Kapitel 3 wurden Fehlerinformationen bislang nicht in diesem Umfang berücksichtigt, um durch dieses Testkriterium eine höhere Aussagekraft der Testdaten zu erzeugen. Ein wesentliches Alleinstellungsmerkmal stellt die Generierung der Testdaten unter Berücksichtigung der zyklischen Ausführungslogik von Bausteinen mit Gedächtnis dar. Mit Hilfe der automatischen Ermittlung der Datenabhängigkeit im Steuerungsquelltext werden Querbezüge in die Analyse integriert und so zeitdiskrete Eingabedaten ermittelt. Durch die Berücksichtigung von Sensor Aktor Relationen können funktionale Zusammenhänge, die häufig durch den technischen Prozess begründet sind, genutzt werden, um die Testtiefe in einem Fehlerszenario entsprechend zu erhöhen.

Das Verfahren ist durch den geringen Modellierungsaufwand und den hohen Automatisierungsgrad bestens für den Einsatz im Maschinen- und Anlagenbau geeignet. Damit kann eine Erhöhung der Absicherung in fehlerbehafteten Situationen erreicht werden. Die zentrale Eigenschaft des Testverfahrens ist die Generierung von Eingabedaten auf Basis des tatsächlich vorhandenen Quelltextes und nicht auf einer Spezifikation. Das Verfahren soll etablierte Black-Box Verfahren nicht ersetzen, sondern muss als orthogonaler Mechanismus verstanden werden. Durch die Betrachtung der tatsächlichen Implementierung wird sichergestellt, dass im Falle des Eintritts eines Fehlers das real vorhandene Steuerungsverhalten erzwungen und somit bewertet werden kann. Dadurch kann u.a. auch sichergestellt werden, ob vermeintlich irrelevanter Quelltext trotzdem zur Ausführung gebracht werden kann.

Der Großteil der existierenden IEC 61131-3 Steuerungsapplikationen ist mit den prozeduralen Paradigma entwickelt worden, weshalb das Konzept diese Situation aufgreift. Die objektorientierten Erweiterungen sind zwar in die Programmiernorm aufgenommen worden, deren praktikable Verwendbarkeit in der Branche muss aber aufgrund der Gegebenheiten erst noch erforscht werden. Die Generierung von Testeingabedaten für zyklisch ausgeführte Quelltexte mit objektorientierten Konzepten erfährt unter Betrachtung von Vererbung, Polymorphie etc. außerdem zusätzliche Komplexitätsgrade.

7.3 Künftige Erweiterungsmöglichkeiten

Der abschließende Abschnitt beschreibt künftige Erweiterungsmöglichkeiten, die das Konzept weiter verfeinern und neue weiterführende Aspekte umfassen.

In den Evaluationsbeispielen wurde die schnelle Berechnung der Testeingabedaten für einen Datensatz demonstriert. Bei der vollständigen Berechnung aller Eingabedaten unterstützt die Eingrenzung der Parameter auf gültige Werte die Performance des Verfahrens positiv. Einen entscheidenden Einfluss auf die Ausführungsdauer haben

insbesondere jene Bedingungen, die ausschließlich aus Variablen, d.h. ohne konkrete Werte bestehen. Dabei ist es unerheblich, welche Vergleichsoperationen (<,<>,=) eingesetzt werden. Eine gesonderte Betrachtung reduziert die Komplexität des gesamten Berechnungsumfangs und erlaubt eine zeitoptimierte Ergebnisermittlung.

In Kapitel 6.4 erfolgte bereits die Betrachtung der Anwendbarkeit des Konzeptes auf die restlichen IEC 61131-3 Programmiersprachen. Durch die Analyse auf einem Zwischenformat wird eine Unabhängigkeit vom Ausgangsquelltext grundsätzlich gefördert, kann jedoch aufgrund der abweichenden Transformationsregeln nicht unmittelbar eingesetzt werden. Um das Konzept der Testdatengenerierung für weitere IEC Programmiersprachen zu unterstützen, müssen die Ausprägungen der Transformation in ST inhärenter Bestandteil der Generierungskette sein.

Fehler können in allen Teilbereichen reaktiver Systeme eintreten und können stets als Abweichung einer Erwartungshaltung betrachtet werden. So kombiniert bspw. die Verfahrenstechnik Elemente der Steuerungstechnik mit kontinuierlichen Prozessabläufen und stellt hohe Anforderungen an die zur Umsetzung benötigten Experten während der Entwicklung und Integration. Fehler, die im technischen Prozess auftreten, können weitreichende Behebungsabläufe nach sich ziehen, so dass bspw. das fehlerhafte Gut abtransportiert und der vollständige Prozess geordnet neu gestartet werden muss. Um derartige Situationen frühzeitig ausschließen und das Systemverhalten diesbezüglich überprüfen zu können, wäre eine Konzepterweiterung um Fehler im technischen Prozess wünschenswert. Diese zeichnen sich jedoch durch große Auswirkungen auf die gesamten Abläufe und Komponenten aus, so dass eine fundierte Untersuchung erforderlich ist.

In dem erarbeiteten Konzept dient das Fehlermodell, neben der Dokumentation des Erkenntnisgewinns, als Grundlage zur Testdatengenerierung. Durch die Integration in das Gesamtsystem stellen diese Modellinformationen eine Präzisierung abnormaler Verhaltensaspekte dar. Durch Simulatoren können Komponenten- und Systemverhaltensbeobachtungen zu Rückschlüssen auf die Stabilität des Systems genutzt werden. Die Eintrittswahrscheinlichkeit von Fehlern ist teilweise sehr niedrig. Zur Simulation der modellierten Fehler im Rahmen des Systemverhaltens werden Simulatoren zur Ausführung der SysML Modelle benötigt, die auch bei sog. *rare events* verlässliche Aussagen ermöglichen.

Nach der Generierung der Testeingabedaten wird für die Ausführung der eigentlichen Testläufe eine Testumgebung benötigt, in der das tatsächliche Systemverhalten verifiziert werden kann. Dazu müssen neben den Eingabedaten auch die erwarteten Ausgabedaten erhoben werden, um die Testläufe mit pass/fail Kriterien bewerten zu können. Eine diffizile Entscheidung bei der Ausführung der Testfälle ist die Wahl der Verifikation gegen eine reale oder virtuelle Anlage/Maschine oder eine hybride Mischform. Dabei spielen Aufwandsabschätzungen und Risikobewertung eine entscheidende Rolle. Diese Fragestellung bedarf einer intensiven Untersuchung, um die optimale Verifikationsstrategie einsetzen zu können.

ANHANG A

Beispielprogramme für den quantitativen Vergleich

A.1 Example01 - Einfache if Struktur

Listing A.1: Example01 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example01
2  VAR_INPUT
3     a: INT;
4  END_VAR
5  VAR_OUTPUT
6     b: INT;
7  END_VAR
8  VAR
9  END_VAR
10
11
12 IF a < 5 THEN
13    b := 1;
14 ELSE
15    b := 0;
16 END_IF
```

Listing A.2: Example01 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6
7  int b;
8  void func(int a)
9  {
10     if ( a  < 5)
11     {
```

```
12          b = 1;
13          klee_assert(0);
14      }
15      else
16      {
17          b = 0;
18      }
19  }
20  int main()
21  {
22      int a0;
23
24      // 1. cycle
25      klee_make_symbolic(&a0, sizeof(int), "a0");
26      func(a0);
27      return 0;
28  }
```

A.2 Example02 - Einfache for Schleife

Listing A.3: Example02 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example02
2  VAR_INPUT
3     a: INT;
4  END_VAR
5  VAR_OUTPUT
6     b: INT;
7  END_VAR
8  VAR
9     i : INT;
10 END_VAR
11
12
13 FOR i := 0 TO 5
14 DO
15    b := a + i;
16    i := i + 1;
17 END_FOR
```

Listing A.4: Example02 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6  int b;
7  void func(int a)
8  {
9      static int i;
10     for (i = 0; i <= 5; i = i + 1)
11     {
12         b = a + i;
13         i = i + 1;
14         klee_assert(0);
15     }
16 }
17 int main()
18 {
19     int a0;
20
21     // 1. cycle
22     klee_make_symbolic(&a0, sizeof(int), "a0");
23     func(a0);
24     return 0;
25 }
```

A.3 Example03 - Einfache while Schleife

Listing A.5: Example03 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example03
2  VAR_INPUT
3    a: INT;
4  END_VAR
5  VAR_OUTPUT
6    b: INT;
7  END_VAR
8  VAR
9    i : INT;
10 END_VAR
11
12
13 i := 0;
14 WHILE i < 5
15 DO
16   b := a + i;
```

```
17    i := i + 1;
18  END_WHILE
```

Listing A.6: Example03 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6
7  int b;
8  void func(int a)
9  {
10     static int i;
11     i = 0;
12     while ( i < 5 )
13     {
14         b = a + i ;
15         i = i + 1;
16         klee_assert(0);
17     }
18 }
19 int main()
20 {
21     int a0;
22
23     // 1. cycle
24     klee_make_symbolic(&a0, sizeof(int), "a0");
25     func(a0);
26     return 0;
27 }
```

A.4 Example04 - Verschachtelte if Strukturen

Listing A.7: Example04 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example04
2  VAR_INPUT
3    a: INT;
4    b: INT;
5    c: INT;
6  END_VAR
7  VAR_OUTPUT
8    d: INT;
9  END_VAR
10 VAR
```

```
11 END_VAR
12
13
14 IF a < 5 THEN
15   IF b > 5 THEN
16     IF c > -2 THEN
17       d := 1;
18     ELSE
19       d := 2;
20     END_IF
21   ELSE
22     d := 3;
23   END_IF
24 ELSE
25   d := 4;
26 END_IF
```

Listing A.8: Example04 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6
7  int d;
8  void func(int a,int b,int c)
9  {
10     if ( a  < 5)
11     {
12         if ( b  > 5)
13         {
14             if ( c  > -2)
15             {
16                 d = 1;
17             }
18             else
19             {
20                 d = 2;
21             }
22         }
23         else
24         {
25             d = 3;
26             klee_assert(0);
27         }
28     }
29     else
```

```
30      {
31          d = 4;
32      }
33  }
34
35  int main()
36  {
37      int a0;
38      int b0;
39      int c0;
40
41      // 1. cycle
42      klee_make_symbolic(&a0, sizeof(int), "a0");
43      klee_make_symbolic(&b0, sizeof(int), "b0");
44      klee_make_symbolic(&c0, sizeof(int), "c0");
45      func(a0,b0,c0);
46      return 0;
47  }
```

A.5 Example05 - Verschachtelte for Schleifen

Listing A.9: Example05 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example05
2  VAR_INPUT
3  END_VAR
4  VAR_OUTPUT
5     a: INT;
6  END_VAR
7  VAR
8     i: INT;
9     j: INT;
10    k: INT;
11 END_VAR
12
13
14 FOR i := 0 TO 5
15 DO
16    FOR j := 0 TO 5
17    DO
18       FOR k := 0 TO 5
19       DO
20          a := a + k - j + i;
21       END_FOR
22    END_FOR
23 END_FOR
```

Listing A.10: Example05 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6
7  int a;
8  void func()
9  {
10     static int i;
11     static int j;
12     static int k;
13     for (i = 0; i  <= 5; i =  i  + 1)
14     {
15         for (j = 0; j  <= 5; j =  j  + 1)
16         {
17             for (k = 0; k  <= 5; k =  k  + 1)
18             {
19                 a = a + k - j + i ;
20                 klee_assert(0);
21             }
22         }
23     }
24 }
25 int main()
26 {
27
28     // 1. cycle
29     func();
30     return 0;
31 }
```

A.6 Example06 - Verschachtelte while Schleifen

Listing A.11: Example06 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example06
2  VAR_INPUT
3    a: INT;
4  END_VAR
5  VAR_OUTPUT
6    b: INT;
7  END_VAR
8  VAR
9    i: INT;
10   j: INT;
```

```
11    k: INT;
12 END_VAR
13
14
15 WHILE i < 5
16 DO
17    WHILE j < 5
18    DO
19       WHILE k < a
20       DO
21          b := b + k - j + i;
22          k := k + 1;
23       END_WHILE
24    j := j + 1;
25    END_WHILE
26 i := i + 1;
27 END_WHILE
```

Listing A.12: Example06 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6
7  int b;
8  void func(int a)
9  {
10     static int i;
11     static int j;
12     static int k;
13     while ( i  < 5)
14     {
15         while ( j  < 5)
16         {
17             while ( k  <  a )
18             {
19                 b =  b  +  k  -  j  +  i ;
20                 k =  k  + 1;
21                 klee_assert(0);
22             }
23             j =  j  + 1;
24         }
25         i =  i  + 1;
26     }
27 }
28 int main()
```

```
29 {
30     int a0;
31 
32     // 1. cycle
33     klee_make_symbolic(&a0, sizeof(int), "a0");
34     func(a0);
35     return 0;
36 }
```

A.7 Example07 - Zyklische Abhängigkeit

Listing A.13: Example07 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example07
2  VAR_INPUT
3    a: INT;
4  END_VAR
5  VAR_OUTPUT
6    d: INT;
7  END_VAR
8  VAR
9    c: INT := 0;
10 END_VAR
11 
12 
13 IF a < 45 THEN
14   IF a = 30 AND c = 1 THEN
15     d := 1;
16   ELSE
17     d := 2;
18   END_IF
19 ELSE
20   c := 1;
21 END_IF
```

Listing A.14: Example07 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6  
7  int d;
8  void func(int a)
9  {
10     static int c;
```

```
11      if ( a  < 45)
12      {
13          if ( a  == 30 &&  c  == 1)
14          {
15              d = 1;
16              klee_assert(0);
17          }
18          else
19          {
20              d = 2;
21          }
22      }
23      else
24      {
25          c = 1;
26      }
27  }
28  int main()
29  {
30      int a0,a1;
31
32      // 1. cycle
33      klee_make_symbolic(&a0, sizeof(int), "a0");
34      func(a0);
35      // 2. cycle
36      klee_make_symbolic(&a1, sizeof(int), "a1");
37      func(a1);
38      return 0;
39  }
```

A.8 Example08 - Doppelte zyklische Abhängigkeit

Listing A.15: Example08 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example08
2  VAR_INPUT
3     a: INT;
4  END_VAR
5  VAR_OUTPUT
6     e: INT;
7  END_VAR
8  VAR
9     b: INT := 0;
10    c: INT := 0;
11    d: INT := 0;
12 END_VAR
13
```

```
14
15 IF a > 180 THEN
16   IF a < 270 AND b = 1 THEN
17     e := 0;
18   ELSE
19     e := 1;
20   END_IF
21 ELSE
22   IF a = 90 AND c = 1 THEN
23     b := 1;
24   ELSE
25     c := 1;
26   END_IF
27 END_IF
```

Listing A.16: Example08 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6
7  int e;
8  void func(int a)
9  {
10   static int b;
11   static int c;
12   static int d;
13   if ( a  > 180)
14   {
15     if ( a  < 270 &&  b  == 1)
16     {
17       e = 0;
18       klee_assert(0);
19     }
20     else
21     {
22       e = 1;
23     }
24   }
25   else
26   {
27     if ( a  == 90 &&  c  == 1)
28     {
29       b = 1;
30     }
31     else
```

```
32      {
33        c = 1;
34      }
35    }
36  }
37
38  int main()
39  {
40  int a0,a1,a2;
41
42  // 1. cycle
43  klee_make_symbolic(&a0, sizeof(int), "a0");
44  func(a0);
45  // 2. cycle
46  klee_make_symbolic(&a1, sizeof(int), "a1");
47  func(a1);
48  // 3. cycle
49  klee_make_symbolic(&a2, sizeof(int), "a2");
50  func(a2);
51  return 0;
52  }
```

A.9 Example09 - Hohe Anzahl Funktionsparameter

Listing A.17: Example09 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example09
2  VAR_INPUT
3    a1: INT;
4    a2: INT;
5    a3: INT;
6    a4: INT;
7    a5: INT;
8    a6: INT;
9    a7: INT;
10   a8: INT;
11   a9: INT;
12   a10: INT;
13   a11: INT;
14   a12: INT;
15   a13: INT;
16   a14: INT;
17   a15: INT;
18   a16: INT;
19 END_VAR
20 VAR_OUTPUT
21   b: INT;
```

```
22 END_VAR
23 VAR
24 END_VAR
25
26
27 IF a1 < 5 THEN
28   b := 1;
29 ELSE
30   b := 0;
31 END_IF
```

Listing A.18: Example09 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6
7  int b;
8  void func(int a1,int a2,int a3,int a4,int a5,int a6,int a7,int a8,
9  int a9,int a10,int a11,int a12,int a13,int a14,int a15,int a16)
10 {
11     if ( a1  < 5)
12     {
13         b = 1;
14         klee_assert(0);
15     }
16     else
17     {
18         b = 0;
19     }
20 }
21 int main()
22 {
23     int a10;
24     int a20;
25     int a30;
26     int a40;
27     int a50;
28     int a60;
29     int a70;
30     int a80;
31     int a90;
32     int a100;
33     int a110;
34     int a120;
35     int a130;
```

```
36      int a140;
37      int a150;
38      int a160;
39
40      // 1. cycle
41      klee_make_symbolic(&a10, sizeof(int), "a10");
42      klee_make_symbolic(&a20, sizeof(int), "a20");
43      klee_make_symbolic(&a30, sizeof(int), "a30");
44      klee_make_symbolic(&a40, sizeof(int), "a40");
45      klee_make_symbolic(&a50, sizeof(int), "a50");
46      klee_make_symbolic(&a60, sizeof(int), "a60");
47      klee_make_symbolic(&a70, sizeof(int), "a70");
48      klee_make_symbolic(&a80, sizeof(int), "a80");
49      klee_make_symbolic(&a90, sizeof(int), "a90");
50      klee_make_symbolic(&a100, sizeof(int), "a100");
51      klee_make_symbolic(&a110, sizeof(int), "a110");
52      klee_make_symbolic(&a120, sizeof(int), "a120");
53      klee_make_symbolic(&a130, sizeof(int), "a130");
54      klee_make_symbolic(&a140, sizeof(int), "a140");
55      klee_make_symbolic(&a150, sizeof(int), "a150");
56      klee_make_symbolic(&a160, sizeof(int), "a160");
57      func(a10,a20,a30,a40,a50,a60,a70,a80,a90,a100,
58           a110,a120,a130,a140,a150,a160);
59      return 0;
60  }
```

A.10 Example10 - Bedingte Eingabeparameter

Listing A.19: Example10 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example10
2  VAR_INPUT
3     a: INT;
4     b: INT;
5  END_VAR
6  VAR_OUTPUT
7     d: INT;
8  END_VAR
9  VAR
10 END_VAR
11
12
13 IF a > b THEN
14    d := 0;
15 ELSE
16    d := 1;
17 END_IF
```

Listing A.20: Example10 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6
7  int d;
8  void func(int a,int b)
9  {
10     if ( a > b )
11     {
12         d = 0;
13         klee_assert(0);
14     }
15     else
16     {
17         d = 1;
18     }
19 }
20 int main()
21 {
22     int a0;
23     int b0;
24
25     // 1. cycle
26     klee_make_symbolic(&a0, sizeof(int), "a0");
27     klee_make_symbolic(&b0, sizeof(int), "b0");
28     func(a0,b0);
29     return 0;
30 }
```

A.11 Example11 - Parameterraumtest mit einem Eingabeparameter

Listing A.21: Example11 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example11
2  VAR_INPUT
3     a: INT;
4  END_VAR
5  VAR_OUTPUT
6     z: BOOL;
7  END_VAR
8  VAR
9  END_VAR
10
11
12 IF a > 0 THEN
13    z := TRUE;
14 END_IF
```

Listing A.22: Example11 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6
7  BOOL z;
8  void func(int a)
9  {
10     if ( a  > 0)
11     {
12         z = 1;
13         klee_assert(0);
14     }
15 }
16 int main()
17 {
18     int a0;
19
20     // 1. cycle
21     klee_make_symbolic(&a0, sizeof(int), "a0");
22     func(a0);
23     return 0;
24 }
```

A.12 Example12 - Parameterraumtest mit zwei Eingabeparametern

Listing A.23: Example12 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example12
2  VAR_INPUT
3     a: INT;
4     b: INT;
5  END_VAR
6  VAR_OUTPUT
7     z: BOOL;
8  END_VAR
9  VAR
10 END_VAR
11
12
13 IF a > 0 AND b > 0 THEN
14    z := TRUE;
15 END_IF
```

Listing A.24: Example12 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6
7  BOOL z;
8  void func(int a,int b)
9  {
10     if ( a  > 0 &&  b  > 0)
11     {
12         z = 1;
13         klee_assert(0);
14     }
15 }
16 int main()
17 {
18     int a0;
19     int b0;
20
21     // 1. cycle
22     klee_make_symbolic(&a0, sizeof(int), "a0");
23     klee_make_symbolic(&b0, sizeof(int), "b0");
24     func(a0,b0);
```

```
25      return 0;
26  }
```

A.13 Example13 - Parameterraumtest mit drei Eingabeparametern

Listing A.25: Example13 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example13
2  VAR_INPUT
3     a: INT;
4     b: INT;
5     c: INT;
6  END_VAR
7  VAR_OUTPUT
8     z: BOOL;
9  END_VAR
10 VAR
11 END_VAR
12
13
14 IF a > 0 AND b > 0 AND c > 0 THEN
15    z := TRUE;
16 END_IF
```

Listing A.26: Example13 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6
7  BOOL z;
8  void func(int a,int b,int c)
9  {
10     if ( a  > 0 &&  b  > 0 &&  c  > 0)
11     {
12         z = 1;
13         klee_assert(0);
14     }
15 }
16 int main()
17 {
18     int a0;
19     int b0;
```

```
20      int c0;
21
22      // 1. cycle
23      klee_make_symbolic(&a0, sizeof(int), "a0");
24      klee_make_symbolic(&b0, sizeof(int), "b0");
25      klee_make_symbolic(&c0, sizeof(int), "c0");
26      func(a0,b0,c0);
27      return 0;
28  }
```

A.14 Example14 - Parameterraumtest mit vier Eingabeparametern

Listing A.27: Example14 - Ausgangsquelltext in ST

```
1  FUNCTION_BLOCK POU_Example14
2  VAR_INPUT
3     a: INT;
4     b: INT;
5     c: INT;
6     d: INT;
7  END_VAR
8  VAR_OUTPUT
9     z: BOOL;
10 END_VAR
11 VAR
12 END_VAR
13
14
15 IF a > 0 AND b > 0 AND c > 0 AND d > 0 THEN
16    z := TRUE;
17 END_IF
```

Listing A.28: Example14 - In C transformierter Quelltext für KLEE

```
1  #include <stdio.h>
2  #include <klee/klee.h>
3  typedef int BOOL;
4  #define true 1
5  #define false 0
6
7  BOOL z;
8  void func(int a,int b,int c,int d)
9  {
10     if ( a  > 0 &&  b  > 0 &&  c  > 0 &&  d  > 0)
11     {
```

```
12          z = 1;
13          klee_assert(0);
14      }
15  }
16  int main()
17  {
18      int a0;
19      int b0;
20      int c0;
21      int d0;
22
23      // 1. cycle
24      klee_make_symbolic(&a0, sizeof(int), "a0");
25      klee_make_symbolic(&b0, sizeof(int), "b0");
26      klee_make_symbolic(&c0, sizeof(int), "c0");
27      klee_make_symbolic(&d0, sizeof(int), "d0");
28      func(a0,b0,c0,d0);
29      return 0;
30  }
```

Literaturverzeichnis

[ABLN06] Andrews, James H.; Briand, Lionel C.; Labiche, Yvan ; Namin, Akbar S.: Using Mutation Analysis for Assessing and Comparing Testing Coverage Criteria. In: *IEEE Trans. Softw. Eng.* 32 (2006), August, Nr. 8, S. 608–624. http://dx.doi.org/10.1109/TSE.2006.83. – DOI 10.1109/TSE.2006.83. – ISSN 0098–5589 (Zitiert auf Seite 58)

[ABRW09] Angermann, Anne; Beuschel, Michael; Rau, Martin ; Wohlfarth, Ulrich: *MATLAB - Simulink - Stateflow: Grundlagen, Toolboxen, Beispiele*. Bd. 6. Oldenbourg Wissenschaftsverlag, 2009 (Zitiert auf Seite 40)

[AERP04] Alyokhin, Vadim; Elbel, Benedikte; Rothfelder, Martin ; Pretschner, Alexander: Coverage Metrics for Continuous Function Charts. In: *Proc. 15th IEEE International Symposium on Software Reliability Engineering* 15 (2004), 11, S. 257–268 (Zitiert auf Seiten 51 und 59)

[AH09] Aichernig, Bernhard K.; He, Jifeng: Mutation testing in UTP. In: *Formal Asp. Comput.* 21 (2009), Nr. 1-2, S. 33–64 (Zitiert auf Seite 55)

[AJ12] Aichernig, Bernhard K.; Jöbstl, Elisabeth: Towards Symbolic Model-Based Mutation Testing: Combining Reachability and Refinement Checking. In: *MBT*, 2012, S. 88–102 (Zitiert auf Seite 58)

[AML11] Andrews, James H.; Menzies, Tim ; Li, Felix C. H.: Genetic Algorithms for Randomized Unit Testing. In: *IEEE Trans. Softw. Eng.* 37 (2011), January, 80–94. http://dx.doi.org/http://dx.doi.org/10.1109/TSE.2010.46. – DOI http://dx.doi.org/10.1109/TSE.2010.46. – ISSN 0098–5589 (Zitiert auf Seite 58)

[APS05] Aichernig, Bernhard K.; Pari Salas, Percy A.: Test Case Generation by OCL Mutation and Constraint Solving. In: *Proceedings of the Fifth International Conference on Quality Software*. IEEE Computer Society (QSIC '05). – ISBN 0–7695–2472–9, 64–71 (Zitiert auf Seiten 57 und 59)

[ASU86] Aho, Alfred V.; Sethi, Ravi ; Ullman, Jeffrey D.: *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986 (Zitiert auf Seite 86)

[Bal97] Balzert, Helmut: *Lehrbuch der Software-Technik, Bd.2, Software-Management, Software-Qualtätssicherung und Unternehmensmodellierung.* Berlin : Spektrum Akademischer Verlag, 1997 (Zitiert auf Seiten 110 und 204)

[Ban09] Banks, Jerry: *Discrete-Event System Simulation.* Pearson, 2009 (Zitiert auf Seite 75)

[BDG04] Baker, Paul ; Dai, Zhen R. ; Grabowski, Jens ; Haugen, Øystein ; Lucio, Serge ; Samuelsson, Eric ; Schieferdecker, Ina ; E, Clay: *The UML 2.0 Testing Profile.* 2004 (Zitiert auf Seite 35)

[BDJ07] Baker, P. ; Dai, Z.R. ; J., Grabowski: *Model-Driven Testing: Using the UML Testing Profile.* Springer, 2007 (Zitiert auf Seite 3)

[Bec12] Beckhoff Automation GmbH: *SPS-Standard IEC 61131- 3.* `http://infosys.beckhoff.com/index.php?content=../content/1031/tcquickstart/html/tcquickstart_iec.htm&id=10622`. Version: September 2012 (Zitiert auf Seite 11)

[Bei90] Beizer, Boris: *Software testing techniques (2nd ed.).* New York, NY, USA : Van Nostrand Reinhold Co., 1990 (Zitiert auf Seiten 51 und 82)

[Ben05] Bender, Klaus: *Embedded Systems - Qualitätsorientierte Entwicklung.* Springer, 2005 (Zitiert auf Seite 62)

[Ber03] Bertram, Matthias: *Der Softwarebug von Hamburg-Altona Berühmt-berüchtigte Softwarefehler.* `http://formal.iti.kit.edu/~beckert/teaching/Seminar-Softwarefehler-SS03/Ausarbeitungen/bertram.pdf`. Version: Sommersemester 2003 (Zitiert auf Seite 30)

[Ber07] Bertolino, Antonia: Software Testing Research: Achievements, Challenges, Dreams. In: *2007 Future of Software Engineering.* Washington, DC, USA : IEEE Computer Society, 2007 (FOSE '07). – ISBN 0–7695–2829–5, S. 85–103 (Zitiert auf Seite 2)

[BGJ09] Bertsche, Bernd ; Göhner, Peter ; Jensen, Uwe ; Schinköthe, Wolfgang ; Wunderlich, Hans-Joachim: *Zuverlässigkeit mechatronischer Systeme: Grundlagen und Bewertung in frühen Entwicklungsphasen.* Springer Berlin Heidelberg, 2009 (Zitiert auf Seiten 33, 34 und 207)

[BKFVH14] Barbieri, Giacomo ; Kernschmidt, Konstantin ; Fantuzzi, Cesare ; Vogel-Heuser, Birgit: A SysML Based Design Pattern for the High-Level Development of Mechatronic Systems to Enhance Re-Usability. In: *IFAC World Congress* 19 (2014), S. 3431–3437 (Zitiert auf Seite 2)

[Bor08] Bork, Tilmann: *Grundlagen Mechanik und Fluidtechnik.* `https://www.zvei.org/fileadmin/user_upload/Fachverbaende/Automation/Veranstaltungen_Messen/ENISO13849/pdf/TB2_3_Bork.pdf`, 2008 (Zitiert auf Seiten 33 und 34)

[BP84] Basili, Victor R. ; Perricone, Barry T.: Software errors and complexity: an empirical investigation0. In: *Commun. ACM* 27 (1984), Januar, Nr. 1, S. 42–52. http://dx.doi.org/10.1145/69605.2085. – DOI 10.1145/69605.2085. – ISSN 0001–0782 (Zitiert auf Seite 17)

[Bro02] Broekman, Bart: *Testing Embedded Software*. Addison-Wesley Longman, 2002 (Zitiert auf Seite 46)

[Bro10] Brost, Michael: Automatisierte Testfallerzeugung auf Grundlage einer zustandsbasierten Funktionsbeschreibung für Kraftfahrzeugsteuergeräte. In: *Schriftenreihe des Instituts für Verbrennungsmotoren und Kraftfahrwesen der Universität Stuttgart*, 2010 (Zitiert auf Seite 58)

[BS08] Burnim, Jacob ; Sen, Koushik: Heuristics for Scalable Dynamic Test Generation / Electrical Engineering and Computer Sciences, University of California at Berkeley. 2008. – Forschungsbericht (Zitiert auf Seite 53)

[BSBF10] Bassi, Luca ; Secchi, Cristian ; Bonfe, Marcello ; Fantuzzi, Cesare: A SysML-Based Methodology for Manufacturing Machinery Modeling and Design. In: *IEEE/ASME Transactions on Mechatronics* 16 (2010), Nr. 6, 1049-1062. http://dx.doi.org/10.1109/TMECH.2010.2073480. – DOI 10.1109/TMECH.2010.2073480 (Zitiert auf Seiten 42, 44 und 69)

[BVH10] Bayrak, Gülden ; Vogel-Heuser, Birgit: Model-Based Development Concept for Hybrid Systems to Support Process Engineers in Thermo-Mechanical Process Development. In: *10th IFAC Workshop on Programmable Devices and Embedded Systems (PDeS)* 10 (2010), S. 179–184 (Zitiert auf Seiten 36 und 38)

[CDE08] Cadar, Cristian ; Dunbar, Daniel ; Engler, Dawson: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for complex Systems Programs. San Diego, CA, USA, December 2008 (Zitiert auf Seiten 53 und 98)

[Cho78] Chow, Tsun S.: Testing Software Design Modeled by Finite-State Machines. In: *Transactions on Software Engineering* SE-4 (1978), Nr. 3, S. 178–187 (Zitiert auf Seite 56)

[Cle10] Cleff, Thorsten: *Basiswissen Testen von Software*. Bd. Auflage 1. W3L GmbH, 2010 (Zitiert auf Seite 30)

[CR04] Cunning, Steven J. ; Rozenblit, Jerzy W.: Automating Test Generation for Discrete Event Oriented Embedded Systems. In: *Journal of Intelligent and Robotic Systems* 41 (2004), S. 87 – 112 (Zitiert auf Seite 58)

[CTF01] Chevalley, P. ; Thevenod-Fosse, P.: Automated generation of statistical test cases from UML state diagrams. In: *Computer Software and Applications Conference, 2001. COMPSAC 2001. 25th Annual International*, 2001, S. 205 –214 (Zitiert auf Seiten 48, 56 und 59)

[Cyg14] Cygnus Solutions: *Cygwin – Get that Linux feeling on Windows.* `http://www.cygwin.com`. Version: 2014 (Zitiert auf Seite 123)

[Dem78] Demillo, R.A.: Hints on Test Data Selection: Help for the Practicing Programmer. In: *Journal Computer magazine* 11 (1978), S. 34–41 (Zitiert auf Seiten 27, 28 und 55)

[Die09] Diegmüller, Frank: *Probabilistische Methode zur Generierung von Stimuli zum Test zustandsbasierter, reaktiver Elektroniksysteme im Automobil*, University of Kassel, Diss., 2009 (Zitiert auf Seite 3)

[Dij61] Dijkstra, Edsger W.: Algol 60 translation : An Algol 60 translator for the x1 and Making a translator for Algol 60 / Mathematisch Centrum, Amsterdam. 1961. – Research Report 35 (Zitiert auf Seite 91)

[DIN85] DIN 55350-31: *Begriffe der Qualitätssicherung und Statistik (DIN 55350-31).* Dezember 1985 (Zitiert auf Seiten 33 und 203)

[DIN95] DIN 1319-1: *Grundlagen der Meßtechnik – Teil 1: Grundbegriffe (DIN 1319-1).* Januar 1995 (Zitiert auf Seite 33)

[DIN02] DIN EN 61508-1: *Funktionale Sicherheit sicherheitsbezogener elektrischer/elektronischer/programmierbarer elektronischer Systeme - Teil 1 (DIN EN 61508-1).* November 2002 (Zitiert auf Seite 30)

[DIN04] DIN EN 61131-1: *Speicherprogrammierbare Steuerungen - Teil 1: Allgemeine Informationen (EN 61131-1:2003).* März 2004 (Zitiert auf Seite 9)

[DIN05] DIN EN ISO 9000: *Qualitätsmanagementsysteme – Grundlagen und Begriffe (ISO 2000:2005).* Dezember 2005 (Zitiert auf Seite 29)

[DIN10] DIN EN 13306: *Instandhaltung – Begriff der Instandhaltung (DIN EN 13306).* Dezember 2010 (Zitiert auf Seite 30)

[Dom07] Dominka, Sven: *Hybride Inbetriebnahme von Produktionsanlagen - von der virtuellen zur realen Inbetriebnahme*, Technische Universität München, Diss., 2007 (Zitiert auf Seiten 13, 15 und 17)

[DRP99] Dustin, Elfriede ; Rashka, Jeff ; Paul, John: *Automated Software Testing.* Addison-Wesley, 1999 (Zitiert auf Seiten 45, 49 und 50)

[DWP07] Deissenboeck, F. ; Wanger, S. ; Pizka, M. ; Teuchert, S. ; Girard, J.-F.: An Activity-Based Quality Model for Maintainability. In: *Proceedings of the 23rd IEEE International Conference on Software Maintenance (ICSM07)*, 2007 (Zitiert auf Seite 70)

[EM12] Estevez, E. ; Marcos, m:: Model-Based Validation of Industrial Control Systems. In: *IEEE Trans. Industrial Informatics* 8 (2012), Nr. 2, S. 302–310 (Zitiert auf Seiten 39 und 44)

[EPL13] EPLAN: *EPLAN Engineering Center*. http://www.eplan.de, 2013 (Zitiert auf Seite 105)

[ERV02] Esparza, Javier ; Römer, Stefan ; Vogler, Walter: An Improvement of McMillan's Unfolding Algorithm. In: *Formal Methods in System Design* 20 (2002), Nr. 3, S. 285–310 (Zitiert auf Seite 47)

[FB04] Favre-Bulle, Bernhard: *Automatisierung komplexer Industrieprozesse: Systeme, Verfahren und Informationsmanagemen*. Springer Vienna, 2004 (Zitiert auf Seite 33)

[FFVH12] Feldmann, Stefan ; Fuchs, Julia ; Vogel-Heuser, Birgit: Modularity, Variant and Version Management in Plant Automation – Future Challenges and State of the Art. In: *Design Conference*. Dubrovnik, Kroatien, 21012, S. 1689–1698 (Zitiert auf Seite 105)

[FL09] Frey, Georg ; Liu, Liu: Modellierung und Simulation vernetzter Automatisierungs- und Regelungssysteme in Modelica. In: *at – Automatisierungstechnik* 57 (2009), Nr. 9, S. 466–476 (Zitiert auf Seiten 40 und 44)

[GÖ2] Gühmann, C: Testautomatisierung und Modellbildung für die Hardware-in-the-Loop Simulation. In: *IIR Tagung Versuch, Test und Simulation*, 2002 (Zitiert auf Seiten 40 und 44)

[GBR98] Gotlieb, Arnaud ; Botella, Bernard ; Rueher, Michel: Automatic test data generation using constraint solving techniques. In: *SIGSOFT Softw. Eng. Notes* 23 (1998), März, Nr. 2, 53–62. `http://dx.doi.org/10.1145/271775.271790`. – DOI 10.1145/271775.271790. – ISSN 0163–5948 (Zitiert auf Seiten 51 und 59)

[GC09] Ghani, Kamran ; Clark, John A.: Automatic Test Data Generation for Multiple Condition and MCDC Coverage. In: *Proceedings of the 2009 Fourth International Conference on Software Engineering Advances*. Washington, DC, USA : IEEE Computer Society, 2009 (ICSEA '09). – ISBN 978–0–7695–3777–1, S. 152–157 (Zitiert auf Seite 58)

[GG75] Goodenough, John B. ; Gerhart, Susan L.: Toward a theory of test data selection. In: *SIGPLAN Not.* 10 (1975), April, Nr. 6, S. 493–510. `http://dx.doi.org/10.1145/390016.808473`. – DOI 10.1145/390016.808473. – ISSN 0362–1340 (Zitiert auf Seite 27)

[GG06] Grochtmann, Matthias ; Grimm, Klaus: Classification trees for partition testing. In: *Software Testing, Verification and Reliability* 3 (2006), S. 63–82 (Zitiert auf Seite 45)

[GHJ94] Gamma, Erich ; Helm, Richard ; Johnson, Ralph. E.: *Design Patterns. Elements of Reusable Object-Oriented Software*. 1st. Amsterdam : Addison-Wesley Longman, 1994 (Zitiert auf Seite 115)

[Gre07] Greifeneder, J.: *Formale Analyse des Zeitverhaltens Netzbasierter Automatisierungssysteme*, Technische Universität Kaiserslautern, Diss., 2007 (Zitiert auf Seite 2)

[Gro10a] Group, Object M.: *Unified Modeling Language: Infrastructure, Version 2.3.* formal, May 2010 (Zitiert auf Seite 35)

[Gro10b] Group, Object M.: *Unified Modeling Language: Superstructure, Version 2.3.* formal, May 2010 (Zitiert auf Seiten 35 und 36)

[GRSV06] Goloubeva, Olga ; Rebaudengo, Maurizio ; Sonza Reorda, Matteo ; Violante, Massimo: *Software-Implemented Hardware Fult Tolerance.* Springer, 2006 (Zitiert auf Seiten 32 und 203)

[Ham13] Hametner, Reinhard: *Test Driven Software Development for Improving the Quality of Control Software for Industrial Automation Systems*, Vienna University of Technology, Austria, Diss., 2013 (Zitiert auf Seite 3)

[Hau06] Hauff, T.: Prozessleitsysteme: Lebenszyklus und Qualität. In: *atp – Automatisierungstechnische Praxis* 48 (2) (2006), S. 34–42 (Zitiert auf Seite 66)

[HKVH11] Hametner, Reinhard ; Kormann, Benjamin ; Vogel-Heuser, Birgit ; Winkler, Dietmar ; Zoitl, Alois: Test Case Generation Approach for Industrial Automation Systems. In: *The 5th International Conference on Automation, Robotics and Applications* ICARA, IEEE, New Zealand, IEEE, 2011, S. 6 (Zitiert auf Seiten 47 und 59)

[HL14] Hamburg, Matthias ; Löwer, Anke: *ISTQB/GTB Standardglossar der Testbegriffe.* September 2014 (Zitiert auf Seite 18)

[HP09] Hagenmeyer, V. ; Piechottka, U.: Innovative Prozessführung – Erfahrungen und Perspektiven. Beitrag anlässlich der beiden Plenarvorträge Innovative Prozessführung – Erfahrungen und Perspektiven. In: *atp – Automatisierungstechnische Praxis* 49 (2) (2009), S. 48–64 (Zitiert auf Seite 66)

[HV05] Hanisch, H.-M. ; Vyatkin, V.: Modeling and Verification of Distributed Control Systems. In: *International Conference "Design, Analysis, and Simulation of Distributed Systems"(DADS).* San Diego, 2005 (Zitiert auf Seiten 16, 26, 39 und 44)

[HWs10] Hametner, Reinhard ; Winkler, Dietmar ; Östreicher, Thomas ; Surnic, Natascha ; Biffl, Stefan: Selecting UML Models for Test-Driven Development along the Automation Systems Engineering Process. In: *Proceedings IEEE Emerging Technologies and Factory Automation (ETFA 2010)*, 2010 (Zitiert auf Seite 70)

[IAY01] Ibrahim, Alaa ; Ammar, Hanry H. ; Yacoub, Sherif M.: A Fault Model for Fault Injection Analysis of Dynamic UML Dynamic Specifications. In: *Proceedings of the 12th International Symposium on Software Reliability Engineering*. Washington, DC, USA : IEEE Computer Society, 2001 (ISSRE '01), S. 74 (Zitiert auf Seite 65)

[IEC02] IEC 61508: *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems*. 2002 (Zitiert auf Seiten 2 und 19)

[IEC03a] IEC 61131-3: *IEC 61131-3 Standard - Programmable controllers - Part 3: Programming languages*. 2. International Electrical Commission, 2003 (Zitiert auf Seiten 10, 85 und 100)

[IEC03b] IEC 61131-8: *IEC 61131-8 Standard - Programmable controllers - Part 8: Guidelines for the application and implementation of programming languages*. 2. International Electrical Commission, 2003 (Zitiert auf Seite 11)

[IEC13] IEC 62061: *Sicherheit von Maschinen – Funktionale Sicherheit sicherheitsbezogener elektrischer, elektronischer und programmierbarer elektronischer Steuerungssysteme*. 2013 (Zitiert auf Seite 2)

[IEE08] IEEE: *IEEE Standard for Software and System Test Documentation (IEEE 829)*. July 2008 (Zitiert auf Seiten 15 und 79)

[Ise07] Isermann, Rolf: *Mechatronische Systeme – Grundlagen*. Springer Berlin Heidelberg, 2007 (Zitiert auf Seiten 33, 34 und 207)

[ISO09] ISO26262: ISO/DIS 26262 - Road vehicles — Functional safety / ISO. Geneva, Switzerland : International Organization for Standardization / Technical Committee 22 (ISO/TC 22), Juli 2009. – Forschungsbericht (Zitiert auf Seite 2)

[ISO11] ISO: *ISO/IEC 9899:2011 Information technology — Programming languages — C*. International Organization for Standardization. – 683 (est.) S. `http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=57853` (Zitiert auf Seite 100)

[ITQ11] ITQ: *VDMA Zukunftsprognose*. `http://www.itq.de/files/itq_corporate_brochure.pdf`. Version: 2011 (Zitiert auf Seiten 8 und 203)

[JH01] Jones, James A. ; Harrold, Mary J.: Test-Suite Reduction and Prioritization for Modified Condition/Decision Coverage. In: *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*. Washington, DC, USA : IEEE Computer Society, 2001 (ICSM '01). – ISBN 0–7695–1189–9, S. 92– (Zitiert auf Seite 58)

[JH11] Jia, Yue ; Harman, Mark: An Analysis and Survey of the Development of Mutation Testing. In: *IEEE Transactions on Software Engineering* 37 (2011), S. 649–678. – ISSN 0098–5589 (Zitiert auf Seite 55)

[JJPB07] Johnson, Thomas A. ; Jobe, Jonathan M. ; Paredis, Christiaan J.J. ; Burkhart, Roger M.: Modeling Continuous System Dynamics in SysML. In: *2007 ASME International Mechanical Engineering Congress and Exposition* American Society of Mechanical Engineers (ASME), 2007, S. IMECE2007–42754 (Zitiert auf Seiten 39 und 44)

[JT09] John, Karl H. ; Tiegelkamp, Michael: *SPS-Programmierung mit IEC 61131-3: Konzepte und Programmiersprachen, Anforderungen an Programmiersysteme, Entscheidungshilfen.* Springer Berlin Heidelberg, 2009 (Zitiert auf Seiten 67 und 207)

[JWAW10] Jöbstl, Elisabeth ; Weiglhofer, Martin ; Aichernig, Bernhard K. ; Wotawa, Franz: When BDDs Fail: Conformance Testing with Symbolic Execution and SMT Solving. In: *Proceedings of the 2010 Third International Conference on Software Testing, Verification and Validation.* Washington, DC, USA : IEEE Computer Society, 2010 (ICST '10). – ISBN 978–0–7695–3990–4, S. 479–488 (Zitiert auf Seiten 52 und 59)

[Kat09] Katzke, Uwe: *Spezifikation und Anwendung einer Modellierungssprache für die Automatisierungstechnik auf Basis der Unified Modeling Language (UML)*, Universität Kassel, Diss., 2009 (Zitiert auf Seite 70)

[KCJ11] Kumar, Barath ; Czybik, Björn ; Jasperneite, Jürgen: Model Based TTCN-3 Testing of Industrial Automation Systems - First results. In: *16th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2011).* Toulouse, France, Sep 2011 (Zitiert auf Seite 58)

[Kel09] Kelley, Kenneth: Automated test case generation from correct and complete system requirements models. In: *IEEE Aerospace conference*, 2009, S. 1–10 (Zitiert auf Seiten 49 und 59)

[KFVH12] Kormann, Benjamin ; Friedrich, Markus ; Vogel-Heuser, Birgit: Befragung deutscher Maschinenbauunternehmen zum Thema Softwaretest – Handlungsbedarf für den Maschinen-/Anlagenbau und Lösungsvorschlag. In: *Tagungsband GMA-Kongress Automation 2012* (2012), Juni, S. 1–12 (Zitiert auf Seiten 2, 12, 14, 15, 27, 28, 36 und 207)

[KHC99] Kim, Y.G. ; Hong, H. S. ; Cho, S.M. ; Bae, D. H. ; Cha, S. D.: Test Cases Generation from UML State Diagrams. In: *IEEE Proceedings: Software* Bd. 146, 1999, S. 187–192 (Zitiert auf Seite 58)

[KHD08] Krause, J. ; Herrmann, A. ; Diedrich, C.: Test case generation from formal system specifications based on UML State Machines. In: *atp - international* (2008) (Zitiert auf Seiten 47 und 59)

[KKBB08] Kock, Peter ; Kuckertz, Christine ; Bienmüller, Tom ; Brockmeyer, Udo: Komponententest von modellbasierten Funktionen im MIL/SIL und PIL. In: *3. Tagung Simulation und Test in der Funktions-*

und Softwareentwicklung für die Automobilelektronik. Berlin, 2008 (Zitiert auf Seiten 38 und 44)

[KS16] Kormann, Benjamin ; Siemers, Christian: Test mechatronischer Systeme – Simulationsparameter für Fehlerszenarien ermitteln. In: *atp – Automatisierungstechnische Praxis* 58 (2016), Nr. 4, S. 44–53 (Zitiert auf Seiten 63, 80 und 143)

[KTVH12] Kormann, Benjamin ; Tikhonov, Dimitry ; Vogel-Heuser, Birgit: Automated PLC Software Testing using adapted UML Sequence Diagrams. In: *14th IFAC Symposium of Information Control Problems in Manufacturing* 14 (2012), May, S. 1–7 (Zitiert auf Seiten 2, 3, 12, 13, 27, 28, 36, 58, 62, 70 und 111)

[KVH11] Kormann, Benjamin ; Vogel-Heuser, Birgit: Automated Test Case Generation Approach for PLC Control Software Exception Handling using Fault Injection. In: *Annual Conference of the IEEE Industrial Electronics Society (IECON)* 37 (2011) (Zitiert auf Seiten 19, 27, 28, 62 und 63)

[KWV10] Kormann, Benjamin ; Witsch, Daniel ; Vogel-Heuser, Birgit: Automatische Testfallgenerierung mittels Model-Checking für Steuerungsprogramme. In: *Tagungsband GMA-Kongress Automation 2010*, 2010 (Zitiert auf Seiten 2, 48 und 59)

[KWW09] Kruse, Peter M. ; Wegener, Joachim ; Wappler, Stefan: A highly configurable test system for evolutionary black-box testing of embedded systems. In: *Proceedings of the 11th Annual conference on Genetic and evolutionary computation*. ACM (GECCO '09). – ISBN 978–1–60558–325–9, 1545–1552 (Zitiert auf Seiten 57 und 59)

[Lap95] Laprie, Jean-Claude: Dependable computing: concepts, limits, challenges. In: *Proceedings of the Twenty-Fifth international conference on Fault-tolerant computing*. Washington, DC, USA : IEEE Computer Society, 1995 (FTCS'95). – ISBN 0–8186–7146–7, S. 42–54 (Zitiert auf Seite 74)

[Lat02] Lattner, Chris: *LLVM: An Infrastructure for Multi-Stage Optimization*. Urbana, IL, Computer Science Dept., University of Illinois at Urbana-Champaign, Diplomarbeit, Dec 2002. – *See* `http://llvm.cs.uiuc.edu`. (Zitiert auf Seite 98)

[LBYF05] Loeis, Kingliana ; Bani Younis, Mohammed ; Frey, Georg: Application of Symbolic and Bounded Model Checking to the Verification of Logic Control Systems. In: *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, 2005 (Zitiert auf Seite 58)

[Leh04] Lehmann, Eckard: *Time partition testing: Systematischer Test des kontinuierlichen Verhaltens von eingebetteten Systemen*, Technische Universität Berlin, Diss., 2004 (Zitiert auf Seiten 19 und 46)

[LFF12] Liu, L ; Felgner, F. ; Frey, G.: Modellierung und Simulation von Cyber-Physical Systems. In: *Proceedings of 12th Fachtagung Entwurf komplexer Automatisierungssysteme (EKA 2012)* 12 (2012), May, S. 149–157 (Zitiert auf Seite 40)

[LG99] Lauber, R. ; Göhner, P.: *Prozessautomatisierung I, 3. Auflage.* Berlin Heidelberg New York : Springer Verlag, 1999 (Zitiert auf Seiten 42 und 69)

[LG10] Lochau, M. ; Goltz, U.: Feature Interaction Aware Test Case Generation for Embedded Control Systems. In: *6. International Workshop on Model-Based Testing - MBT 2010, Paphos, Cyprus* 6 (2010) (Zitiert auf Seite 3)

[LI07] Lefticaru, Raluca ; Ipate, Florentin: Automatic State-Based Test Generation Using Genetic Algorithms. In: *Proceedings of the Ninth International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*. IEEE Computer Society. – ISBN 0–7695–3078–8, 188–195 (Zitiert auf Seiten 56 und 59)

[Lin08] Lindner, Paul: *Constraintbasierte Testdatenermittlung für Automatisierungssoftware auf Grundlage von Signalflussplänen*, Universität Stuttgart, Diss., 2008 (Zitiert auf Seiten 3, 55 und 59)

[Liu14] Liu, Liu: *Object-oriented Modeling and Efficient Simulation of C3-Systems*, Universität des Saarlandes, Saarbrücken, Germany, Diss., Jan 2014 (Zitiert auf Seite 39)

[log09] logi.DIAG: *Test Driven Automation und Conditioning Monitoring in der Systemumgebung logi.cals.* 2009 (Zitiert auf Seiten 54 und 59)

[LSP82] Lamport, Leslie ; Shostak, Robert ; Pease, Marshall: The Byzantine Generals Problem. In: *ACM Trans. Program. Lang. Syst.* 4 (1982), Juli, Nr. 3, S. 382–401. – ISSN 0164–0925 (Zitiert auf Seite 31)

[LW95] Lai, Ming-Yee ; Wang, Steve Y.: Software fault insertion testing for fault tolerance. In: *Software Fault-Tolerance, MR Lyu Editor* (1995), S. 315–333 (Zitiert auf Seite 74)

[LW05] Lammermann, Frank ; Wappler, Stefan: Benefits of software measures for evolutionary white-box testing. In: *Proceedings of the 2005 conference on Genetic and evolutionary computation.* New York, NY, USA : ACM, 2005 (GECCO '05). – ISBN 1–59593–010–8, S. 1083–1084 (Zitiert auf Seite 16)

[MB03] Marré, Martina ; Bertolino, Antonia: Using Spanning Sets for Coverage Testing. In: *IEEE Trans. Softw. Eng.* 29 (2003), November, Nr. 11, S. 974–984. http://dx.doi.org/10.1109/TSE.2003.1245299. – DOI 10.1109/TSE.2003.1245299. – ISSN 0098–5589 (Zitiert auf Seite 58)

[MBLDP11] Mouchawrab, Samar ; Briand, Lionel C. ; Labiche, Yvan ; Di Penta, Massimiliano: Assessing, Comparing, and Combining State Machine-Based Testing and Structural Testing: A Series of Experiments. In: *IEEE Trans. Softw. Eng.* 37 (2011), März, Nr. 2, S. 161–187. http://dx.doi.org/10.1109/TSE.2010.32. – DOI 10.1109/TSE.2010.32. – ISSN 0098–5589 (Zitiert auf Seite 50)

[McC09] McConnell, Steve: *Code Complete*. Bd. 2. Microsoft Press, 2009 (Zitiert auf Seiten 16 und 203)

[McM04] McMinn, Phil: Search-based software test data generation: a survey. In: *Software Testing, Verification and Reliability* 14 (2004), Nr. 2, S. 105–156. http://dx.doi.org/10.1002/stvr.294. – DOI 10.1002/stvr.294. – ISSN 1099–1689 (Zitiert auf Seite 3)

[MF01] Mertke, Thomas ; Frey, Georg: Formal Verification of PLC-Programs Generated from Signal Interpreted Petri Nets. In: *Proceedings of the 2001 IEEE Systems, Man, and Cybernetics Conference*. Springer-Verlag (ICATPN'03). – ISBN 3–540–40334–5, 440–449 (Zitiert auf Seite 58)

[Min11] Minion: *Minion 0.12 – Fast, Scalable Constraing Solving*. http://minion.sourceforge.net, March 2011 (Zitiert auf Seite 84)

[MK09] Majchrzak, Tim A. ; Kuchen, Herbert: Automated Test Case Generation Based on Coverage Analysis. In: *Proceedings of the 2009 Third IEEE International Symposium on Theoretical Aspects of Software Engineering*. Washington, DC, USA : IEEE Computer Society, 2009 (TASE '09). – ISBN 978–0–7695–3757–3, S. 259–266 (Zitiert auf Seiten 51 und 59)

[Mod10] Modelica Association: *Modelica® - A Unified Object-Oriented Language for Physical Systems Modeling, Language Specification*. Bd. Version 3.2. Linköping, Schweden : MODELICA ASSOCIATION, 2010 (Zitiert auf Seite 39)

[Mon01] Montenegro, Sergio: Methoden und Techniken für die Entwicklung sicherheitsrelevanter Systeme / GMD-FIRST. 2001. – Forschungsbericht (Zitiert auf Seiten 2 und 65)

[MP91] Meyer, F. J. ; Pradhan, D. K.: Consensus With Dual Failure Modes. In: *IEEE Trans. Parallel Distrib. Syst.* 2 (1991), April, Nr. 2, S. 214–222. – ISSN 1045–9219 (Zitiert auf Seite 31)

[Muk08] Mukherjee, Shubu: *Architecture Design for Soft Errors*. Morgan Kaufmann, 2008 (Zitiert auf Seiten 32 und 203)

[MW10] Mohacsi, S.; Wallner, J.: A Hybrid Approach for Model-Based Random Testing. In: *Advances in System Testing and Validation Lifecycle (VALID), 2010 Second International Conference on Advances in System Testing and Validation Lifecycle*, 2010, S. 10–15 (Zitiert auf Seiten 54 und 59)

[MWD05] Marsal, G.; Witsch, D.; Denis, B.; Faure, J.-M. ; Frey, G.: Evaluation of Real-Time Capabilities of Ethernet-based Automation Systems using Formal Verification and Simulation. In: *Proceedings of 1ère Rencontres des Jeunes Chercheurs en Informatique Temps Réel*. Nancy, France, September 2005, S. 27–30 (Zitiert auf Seite 58)

[Mye79] Myers, Glenford J.: *The Art of Software Testing*. Verlag John Wiley & Sons, Inc, 1979 (Zitiert auf Seiten 19, 28 und 50)

[Nta88] Ntafos, Simeon C.: A Comparison of Some Structural Testing Strategies. In: *IEEE Trans. Softw. Eng.* 14 (1988), Juni, Nr. 6, S. 868–874. `http://dx.doi.org/10.1109/32.6165`. – DOI 10.1109/32.6165. – ISSN 0098–5589 (Zitiert auf Seite 50)

[NWM02] Netisopakul, P.; White, L. J. ; Morris, J.: Data coverage testing. In: *Software Engineering Conference, 2002. Ninth Asia-Pacific*, 2002, S. 465–472 (Zitiert auf Seiten 53 und 59)

[OB88] Ostrand, T. J.; Balcer, M. J.: The category-partition method for specifying and generating fuctional tests. In: *Commun. ACM* 31 (1988), Juni, Nr. 6, S. 676–686. `http://dx.doi.org/10.1145/62959.62964`. – DOI 10.1145/62959.62964. – ISSN 0001–0782 (Zitiert auf Seite 46)

[OMG06] OMG MOF: *MOF Core Specification 2.0*. Januar 2006 (Zitiert auf Seite 35)

[OMG10] OMG: *OMG Systems Modeling Language (OMG SysML) v1.2*. June 2010 (Zitiert auf Seiten 36, 37 und 203)

[OMG12] OMG UTP: *UML Testing Profile 1.1*. April 2012 (Zitiert auf Seite 35)

[Ott08] Otto, Andreas: Der Weg zum sicheren Funktionsbaustein. In: *SICHERE AUTOMATION* SPS-Special (2008), S. 254 – 256 (Zitiert auf Seite 58)

[PEN07] PENTASYS AG: *QS-Report 2007*. Pentasys, 2007 (Zitiert auf Seite 13)

[Pfa10] Pfaller, Christian: *Anforderungsorientierter modellbasierter Softwaretest reaktiver Systeme*, TU München, Diss., 2010 (Zitiert auf Seite 58)

[PJR09] Page, A.; Johnston, K. ; Rollison, B.: *How We Test Software At Microsoft*. Microsoft Press, 2009 (Zitiert auf Seiten 2 und 62)

[PM10] Papadakis, Mike ; Malevris, Nicos: Automatic Mutation Test Case Generation via Dynamic Symbolic Execution. In: *Proceedings of the 2010 IEEE 21st International Symposium on Software Reliability Engineering*. Washington, DC, USA : IEEE Computer Society, 2010 (ISSRE '10). – ISBN 978–0–7695–4255–3, S. 121–130 (Zitiert auf Seiten 56 und 59)

[RBJ00] Rusu, Vlad ; Bousquet, Lydie d. ; Jéron, Thierry: An Approach to Symbolic Test Generation. In: *Proceedings of the Second International Conference on Integrated Formal Methods*. London, UK, UK : Springer-Verlag, 2000 (IFM '00). – ISBN 3–540–41196–8, S. 338–357 (Zitiert auf Seite 52)

[REG13] Reinhart, G. ; Engelhardt, P ; Geiger, F. ; Philipp, T. ; Wahlster, W. ; Zühlke, D. ; Schlick, J. ; Becker, T. ; Löckelt, M. ; Prirvu, B. ; Stephan, P. ; Hodek, S. ; Scholz-Reiter, B. ; Thoben, K. ; Gorldt, C. ; Hribernik, K. ; Lappe, D. ; Veigt, M.: Cyber-Physische Produktionssysteme – Produktivitäts- und Flexibilitätssteigerung durch die Vernetzung intelligenter Systeme in der Fabrik. In: *wt-online* Ausgabe 2-2013 SONDERHEFT INDUSTRIE 4.0 (2013), S. 84–89 (Zitiert auf Seite 1)

[Rin02] Rink, Anton W.: *Entwicklung einer Methode für die systemtechnische Auslegung verteilter und sicherheitskritischer Führungsfunktionen für Fahrzeugantriebe*, Bergische Universität Wuppertal, Diss., 2002 (Zitiert auf Seiten 41 und 44)

[RN10] Russell, Stuart J. ; Norvig, Peter: *Artificial Intelligence: A Modern Approach*. Bd. 3. Prentice Hall, 2010 (Zitiert auf Seite 84)

[RRTVH14] Rösch, Susanne ; Reiterer, Bernhard ; Tikhonov, Dmitry ; Vogel-Heuser, Birgit: Modellbasierter Fehlerinjektions-Applikationstest für SPS-Programme basierend auf dem CODESYS Test Manager. In: *SPS/IPC/Drives*. Nürnberg, 2014 (Zitiert auf Seite 3)

[RS03] Rosen, J.P. ; Strohmeier, A.: Reliable Software Technologies - Ada-Europe 2003, 8th Ada-Europe International Conference on Reliable Software Technologies. In: *Ada-Europe* Bd. 2655. Toulouse, France : Springer, June 2003 (Zitiert auf Seiten 30, 31 und 203)

[RTSVH14] Rösch, Susanne ; Tikhonov, Dimitry ; Schütz, Daniel ; Vogel-Heuser, Birgit: Model-based testing of PLC software: test of plants' reliability by using fault injection on component level. In: *IFAC World Congress (IFAC 2014)* 19 (2014), S. 3509–3515 (Zitiert auf Seite 3)

[RUPVH15] Rösch, S. ; Ulewicz, S. ; Provost, J. ; Vogel-Heuser, B.: Review of Model-Based Testing Approaches in Production Automation and Adjacent Domains – Current Challenges and Research Gaps. In: *Journal of Software Engineering and Applications* 8 (2015), S. 499–519 (Zitiert auf Seite 4)

[Rus13] *Kapitel* Software: Die Zukunft der Industrie. In: Russwurm, Siegfried: *Beherrschung der industriellen Komplexität mit SysLM*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2013. – ISBN 978–3–642–36917–9, S. 21–36 (Zitiert auf Seite 1)

[Sch99] Schneider, Hans-Josef: Diagnosesystem für Feldgeräte – Stand der Anwenderforderungen. In: *atp – Automatisierungstechnische Praxis* 41 (1999), Nr. 2, S. 40–48 (Zitiert auf Seiten 34 und 203)

[Sch05] Scholz, Peter: *Softwareentwicklung, eingebetteter Systeme: Grundlagen, Modellierung, Qualitätssicherung*. Springer, 2005 (Zitiert auf Seite 38)

[Sch06] Schuler, H.: Automation in Chemical Industry. In: *at – Automatisierungstechnik* 8 (2006), S. 363–371 (Zitiert auf Seite 66)

[Sch07] Scheler, Fabian: *Fehlertypen – eine kurze Übersicht*. Vorlesungsfolien "Fehlertoleranz in verteilten Systemen und Echtzeitsystemen". `https://www4.cs.fau.de/z/Lehre/SS07/HS_FVSEZS/Themen/Fehlertypen.pdf`. Version: 2007 (Zitiert auf Seiten 32 und 203)

[SEK09] Seitz, M. ; Ehret, V. ; Kiefer, M. ; Ziegler, A. ; Kruschitz, E. ; Usselmann, E.: Automatisches Testen von Automatisierungssystemen. In: *http://www.automatisierungs-region.de* (2009) (Zitiert auf Seite 58)

[SF09] Soliman, D. ; Frey, G.: Verification and Validation of Safety Applications based on PLCopen Safety Function Blocks using Timed Automata in Uppaal. In: Fanti, Mariagrazia (Hrsg.): *2nd IFAC workshop on Dependable Control of Discrete Systems*. École Normale Supérieure de Cachan, France, 2009 (Zitiert auf Seite 58)

[SF12] Soliman, D. ; Frey, G.: Function Block Diagram to UPPAAL Timed Automata Transformation Based on Formal Models. In: *14th IFAC Symposium on Information Control Problems in Manufacturing (INCOM 2012), Bucharest, Romania* W (2012), S. 625–631 (Zitiert auf Seite 2)

[SLVH09] Sim, T. Y. ; Li, F. ; Vogel-Heuser, B.: Benefits of an Interdisciplinary Modular Concept in Automation of Machine and Plant Manufacturing. In: *13th IFAC Symposium on Information Control Problems in Manufacturing (INCOM)* 13 (2009), S. 898–903 (Zitiert auf Seite 26)

[SP10] Schwarzl, Christian ; Peischl, Bernhard: Test Sequence Generation from Communicating UML State Charts: An Industrial Application of Symbolic Transition Systems. In: *Proceedings of the 2010 10th International Conference on Quality Software*. Washington, DC, USA : IEEE Computer Society, 2010 (QSIC '10). – ISBN 978–0–7695–4131–0, S. 122–131 (Zitiert auf Seiten 48 und 59)

[STF11] Soliman, D.; Thramboulidis, K. ; Frey, G.: A methodology to upgrade legacy industrial systems to meet safety regulations. In: *Dependable Control of Discrete Systems (DCDS), 2011 3rd International Workshop on*, 2011, S. 141 –147 (Zitiert auf Seite 42)

[SV05] Schlingloff, Bernd-Holger; Vulinovic, Sasa: Zuverlässigkeitsprüfung eingebetteter Steuergeräte mit modellgetriebener Fehlerinjektion. In: *Proceedings der Jahrestagung der ASIM/GI-Fachgruppe 4.5.5* Simulation technischer Systeme (2005). `http://edoc.hu-berlin.de/docviews/abstract.php?id=26387`. – [Online: Stand 2010-08-04T14:28:31Z] (Zitiert auf Seite 65)

[SV07] Stahl, T.; Voelters, M.: *Modellgetriebene Softwareentwicklung*. Bd. 2nd. dpunkt.verlag, 2007 (Zitiert auf Seite 35)

[SW09] Schütz, Daniel; Wannagat, Andreas: Domänenspezifische Modellierung für automatisierungstechnische Anlagen mit Hilfe von SysML. In: *ATP* 03 (2009) (Zitiert auf Seiten 26, 42, 44 und 69)

[SZC08] Sünder, Christoph K.; Zoitl, Alois; Christensen, James H.; Steininger, Heinrich ; Fritsche, Josef: Considering IEC 61131-3 and IEC 61499 in the context of Component Frameworks. In: *Proceedings IEEE INDIN 2008*, 2008, S. 277–282 (Zitiert auf Seite 27)

[Thr05] Thramboulidis, Kleanthis: IEC 61499 in Factory Automation. In: *Proceedings of the International Conference on Industrial Electronics, Technology and Automation*, 2005 (Zitiert auf Seite 27)

[Thr10] Thramboulidis, Kleanthis: The 3+1 SysML View-Model in Model Integrated Mechatronics. In: *Journal of Software Engineering and Applications (JSEA)* 3 (2010), Nr. 2, S. 109–118 (Zitiert auf Seiten 42, 44 und 69)

[TKVH12] Thoma, Andreas; Kormann, Benjamin ; Vogel-Heuser, Birgit: Fault-Centric System Modeling using SysML for Reliability Testing. In: *IEEE Conference on Emerging Technologies and Factory Automation (ETFA)* 17th (2012), S. 1–8 (Zitiert auf Seiten 63 und 69)

[TP88] Thambidurai, Philip M.; Park, You-Keun: Interactive Consistency with Multiple Failure Modes. In: *SRDS*, 1988, S. 93–100 (Zitiert auf Seite 31)

[Try16] TrySim: *TrySim – 3D Maschinen-Simulation*. `http://www.trysim.de/de/schnittstellen/`. Version: Mai 2016 (Zitiert auf Seiten 114 und 143)

[TSF11] Thramboulidis, Kleanthis; Soliman, Doaa ; Frey, Georg: Towards an automated verification process for industrial safety applications. In: *CASE*, 2011, S. 482–487 (Zitiert auf Seite 58)

[UL07] Utting, Mark ; Legeard, Bruno: *Practical Model-based Testing.* Morgan Kaufmann, 2007. – 456 S (Zitiert auf Seite 3)

[USVH14] Ulewicz, Sebastian ; Schütz, Daniel ; Vogel-Heuser, Birgit ; Korajda, Bartosz ; Hess, Dieter: Modellbasierte Auswirkungsbewertung von Änderungen und Testanpassung für SPS-Steuerungssoftware im Maschinen- und Anlagenbau. In: *Automation 2014.* Baden-Baden, Deutschland, Jul 2014 (Zitiert auf Seite 3)

[VB08] Vyatkin, V. ; Bouzon, G.: Using visual specifications in verification of industrial automation controllers. In: *EURASIP J. Embedded Syst.* 8 (2008), S. 1–9. – ISSN 1687–3955 (Zitiert auf Seite 39)

[VH05] Vyatkin, Valeriy ; Hanisch, Hans-Michael: Verification of distributed control systems in intelligent manufacturing. In: *Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation (ETFA).* Catania, Italy, September 2005. – ISSN 0950–5849 (Zitiert auf Seite 58)

[VHBKF11] Vogel-Heuser, Birgit ; Braun, Steven ; Kormann, Benjamin ; Friedrich, David: Implementation and evaluation of UML as modeling notation in object oriented software engineering for machine and plant automation. In: *IFAC World Congress*, 2011 (Zitiert auf Seiten 38 und 71)

[VHBO11] Vogel-Heuser, Birgit ; Braun, Steven ; Obermeier, Martin ; Sommer, Kerstin ; Seidel, Tina ; Johannes, Christine: Modeling order effects on errors in object oriented modeling for machine and plant automation from an educational point of view. In: *ETFA*, 2011, S. 1–4 (Zitiert auf Seite 32)

[VHBO12] Vogel-Heuser, Birgit ; Braun, Steven ; Obermeier, Martin ; Jobst, F. ; Schweizer, K.: Usability Evaluation on Teaching and Applying Model-Driven Object Oriented Approaches for PLC Software. In: *American Control Conference (ACC)*, 2012 (Zitiert auf Seite 36)

[VHFH12] Vogel-Heuser, B. ; Frey, G. ; Hermanns, H. ; Renzhin, D. ; Folmer, J. ; Liu, L. ; Hartmanns, A.: Modeling of Networked Automation Systems for Simulation and Model Checking of Time Behavior. In: *IEEE Conference on Systems, Analysis and Automatic Control (SAC'12)* 9 (2012) (Zitiert auf Seite 2)

[VHS11] Vogel-Heuser, B. ; Sommer, K.: A methodological approach to evaluate the benefit and usability of different modeling notations for automation systems. In: *IEEE 7th International Conference on Automation Science and Engineering* 7 (2011), S. 474–481 (Zitiert auf Seite 26)

[VHSFL14] Vogel-Heuser, Birgit ; Schütz, Daniel ; Frank, Timo ; Legat, Christoph: Model-driven engineering of Manufacturing Automation Software Projects – A SysML-based approach. In: *Mechatronics* (2014) (Zitiert auf Seite 2)

[VHSK07] Vogel-Heuser, B. ; Sim, T.Y. ; Katzke, U. ; Wannagat, A. ; Jochem, R.: Evaluation und Anwendung von Variantenmodellierung im Maschinen- und Anlagenbau zur Verbesserung der Modellstruktur und Erhöhung der Wiederverwendung. In: *Gesellschaft Mess- und Automatisierungstechnik (GMA)-Kongress*. Düsseldorf, 2007, S. 107–118 (Zitiert auf Seite 105)

[VJND11] Vudatha, C.P. ; Jammalamadaka, S.K.R. ; Nalliboena, S. ; Duvvuri, B.K.K.: Automated generation of test cases from output domain and critical regions of embedded systems using genetic algorithms. In: *2nd National Conference on Emerging Trends and Applications in Computer Science* 2 (2011), S. 1–6 (Zitiert auf Seiten 57 und 59)

[VS05] Vulinovic, S. ; Schlingloff, H.: Model Based Dependability Evaluation for Automotive Control Functions. In: Fritzson, P. (Hrsg.): *Modeling and Simulation for Public Safety, Conference on*. Linköping, 2005 (Zitiert auf Seiten 41 und 44)

[War09] Wardana, Awang Noor Indra: *Development of Automatic Program Verification for Continuous Function Chart based on Model Checking*, Universität Kassel, Diss., 2009 (Zitiert auf Seite 2)

[WBD10] Wagner, S. ; Broy, M. ; Deissenböck, F. ; Kläs, M. ; Liggesmeyer, P. ; Münch, J. ; Streit, J.: Softwarequalitätsmodelle. Praxisempfehlungen und Forschungsagenda. In: *Informatik-Spektrum* 33 (2010), Nr. 1, S. 37–44 (Zitiert auf Seite 70)

[Wei08] Weilkiens, Tim: *Systems Engineering with SysML/UML: Modeling, Analysis, Design*. Morgan Kaufmann, 2008 (Zitiert auf Seite 72)

[Wün07] Wünsch, Georg: *Methoden für die virtuelle Inbetriebnahme automatisierter Produktionssysteme*, TU München, Diss., 2007 (Zitiert auf Seiten 13 und 17)

[WRK10] Witsch, Daniel ; Ricken, Maria ; Kormann, Benjamin: PLC State Charts: An Approach to Integrate UML State Charts in Open-Loop Control Engineering. In: *IEEE Information Technologies for Sustainable Development*, 2010 (Zitiert auf Seiten 35, 38, 65, 82 und 112)

[WS08] Weissleder, Stephan ; Schlingloff, Bernd-Holger: Models in Software Engineering. Berlin, Heidelberg : Springer-Verlag, 2008, Kapitel Deriving Input Partitions from UML Models for Automatic Test Generation, S. 151–163. – ISBN 978–3–540–69069–6 (Zitiert auf Seiten 47 und 59)

[WSVH08] Witsch, Daniel ; Schünemann, Ulf ; Vogel-Heuser, Birgit: Steigerung der Effizienz und Qualität von Steuerungsprogrammen durch Objektorientierung und UML. In: *atp* (2008), November, S. pp. 42 – 47 (Zitiert auf Seiten 35, 65 und 112)

[WVH11] Witsch, Daniel; Vogel-Heuser, Birgit: PLC-Statecharts: An Approach to Integrate UML-Statecharts in Open-Loop Control Engineering – Aspects on Behavioral Semantics and Model-Checking. In: *Proceedings of 18th IFAC World Congress* 8 (2011) (Zitiert auf Seiten 38, 65, 71 und 82)

[WVJF06] Witsch, Daniel; Vogel-Heuser, Birgit ; Jean-Marc Faure, Gaelle M.: Performance Analysis of Industrial Ethernet Networks by Means of Timed Model-Checking. In: *IFAC INCOM*, 2006 (Zitiert auf Seite 2)

[Zan06] Zankl, Arnold: *Meilensteine der Automatisierung. Vom Transistor zur Digitalen Fabrik*. Bd. 1. Publicis Publishing, 2006 (Zitiert auf Seiten 1, 8, 10 und 203)

[ZSL08] Zäh, Michael F.; Spitzweg, Michael ; Lacour, Frederic-Felix: Einsatz eines Physikmodells zur Simulation des Materialflusses einer Produktionsanlage. In: *it – Information Technology Bd. 50* Oldenbourg Wissenschaftsverlag GmbH, 2008, S. 192–198 (Zitiert auf Seiten 43 und 44)

[ZSSB09] Zoitl, Alois; Strasser, Thomas; Sünder, Christoph ; Baier, Thomas: Is IEC 61499 in Harmony with IEC 61131-3. In: *IEEE Industrial Electronics Magazine* 3 (2009), Nr. 4, S. 49–55 (Zitiert auf Seite 27)

[ZuM15] ZuMaTra: Schlussbericht zu dem IGF-Vorhaben: Steigerung der Zuverlässigkeit von Maschinen und Anlagen durch automatisiertes Testen von Fehlerbehandlungsroutinen in der Steuerungssoftware. 2015. – Forschungsbericht (Zitiert auf Seiten 49 und 59)

[ZY01] Zhang, Yuanlin; Yap, Roland H. C.: Making AC-3 an optimal algorithm. In: *In Proceedings of 17th International Joint Conference on Artificial Intelligence (IJCAI)*, 2001, S. 316–321 (Zitiert auf Seite 84)

Abbildungsverzeichnis

Tabellenverzeichnis